10-2016

Techniques for identifying mobile platform vulnerabilities and detecting policy-violating applications

Mon Kywe SU
Singapore Management University, monkywe.su.2011@phdis.smu.edu.sg

Techniques for Identifying Mobile Platform Vulnerabilities and Detecting Policy-Violating Applications

by

Su Mon Kywe

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Singapore Management University
2016

Techniques for Identifying Mobile Platform Vulnerabilities and Detecting Policy-Violating Applications

Su Mon Kywe

Abstract

Mobile systems are generally composed of three layers of software: application layer where third-party applications are installed, framework layer where Application Programming Interfaces (APIs) are exposed, and kernel layer where low-level system operations are executed. In this dissertation, we focus on security and vulnerability analysis of framework and application layers. Security mechanisms, such as Android's sandbox and permission systems, exist in framework layer, while malware scanners protects application layer. However, there are rooms for improvement in both mechanisms. For instance, Android's permission system is known to be implemented in ad-hoc manner and not well-tested for vulnerabilities. Application layer also focuses mainly on malware application detection, while different types of harmful applications exist on application markets. This dissertation aims to close these security gaps by performing vulnerability analysis on mobile frameworks and detecting policy-violating applications. As a result of our analysis, we find various framework-level vulnerabilities and we are able to launch serious proof-of-concept attacks on both iOS and Android platforms. We also propose mechanisms for detecting policy-violating applications and camouflaged applications. Our techniques are shown to improve the security of mobile systems and have several impacts on mobile industry.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Associate Professor Li Yingjiu, Professor Deng Robert, Associate Professor Ding Xuhua, and Doctor Li Tieyan for their guidance in completing my dissertation.

I am also very grateful to Associate Professor Jason Hong, Professor Lorrie Faith Cranor and Associate Professor Patrick Tague for their mentor-ship during my exchange program in Carnegie Mellon University.

I am also thankful to Professor Peng Ning, Doctor Zhang Xin Wen, Doctor Michael Grace, and Doctor Kunal Petal for their advices and feedbacks during my internship in Samsung Research America.

I also thank my fellow post doctorate researchers, research engineers and research assistants for their research collaboration and suggestions, as well as my seniors, classmates and juniors for their friendship and their encouragement.

Finally, I would like to thank my family and my friends for always supporting me and encouraging me with their best wishes.

Dedication

I dedicate my dissertation work to my parents, Hoke Sein and Mya Mya Win.

List of Publications

Conference Papers

Su Mon Kywe, Yingjiu Li, Kunal Petal, and Michael Grace: Attacking Android Smartphone Systems without Permissions. The 14th International Conference on Privacy, Security and Trust (PST 2016), Auckland, New Zealand, December 12-14, 2016.

Su Mon Kywe, Yingjiu Li, Jason Hong, and Yao Cheng: Dissecting Developer Policy Violating Apps: Characterization and Detection. The 11th International Conference on Malicious and Unwanted Software (Malcon 2016), Fajardo, Puerto Rico, USA, October 11-14, 2016.

Chandrasekhar Bhagavatula, Blase Ur, Kevin Iacovino, **Su Mon Kywe**, Lorrie Faith Cranor, and Marios Savvides: Biometric Authentication on iPhone and Android: Usability, Perceptions, and Influences on Adoption. The Workshop on Usable Security (USEC-2015), San Diego, California, February 8, 2015.

Su Mon Kywe, Yingjiu Li, Robert H. Deng, and Jason Hong: Detecting Camouflaged Applications on Mobile Application Markets. The 17th Annual International Conference on Information Security and Cryptology (ICISC-2014), Seoul, Korea, December 3-5, 2014.

Su Mon Kywe, Christopher Landis, Yutong Pei, Justin Satterfield, Yuan Tian, and Patrick Tague: Extending Private Browsing Mode to Android Mobile Applications. The 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communication (TrustCom-2014), Beijing, China, September 24-26, 2014.

Jie Shi, **Su Mon Kywe**, and Yingjiu Li: Batch Clone Detection for RFID-enabled Supply Chain. The 8th Annual IEEE International Conference on RFID (IEEE RFID 2014), Orlando, Florida, USA, April 8-10, 2014.

Su Mon Kywe, Yingjiu Li, and Jie Shi: Attack and Defense Mechanisms of Malicious EPC Event Injection in EPC Discovery Service. The 2013 IEEE International Conference on RFID Technologies and Applications (IEEE RFID TA), Johor Bahru, Malaysia, September 4-5, 2013.

Jin Han, **Su Mon Kywe**, Qiang Yan, Feng Bao, Robert H. Deng, Debin Gao, Yingjiu Li, and Jianying Zhou: Launching Generic Attacks on iOS with Approved Third-Party Applications. The 11th International Conference on Applied Cryptography and Network Security (ACNS 2013), Banff, Alberta, Canada, June 25-28, 2013.

Su Mon Kywe, Ee-Peng Lim, and Feida Zhu: A Survey of Recommender Systems in Twitter. The 4th International Conference on Social Informatics (SocInfo 2012), Lausanne, Switzerland, December 5-7, 2012.

Su Mon Kywe, Tuan-Anh Hoang, Ee-Peng Lim, and Feida Zhu: On Recommending Hashtags in Twitter Networks. The 4th International Conference on Social Informatics (SocInfo 2012), Lausanne, Switzerland, December 5-7, 2012

Journal Papers

Su Mon Kywe, Jie Shi, Yingjiu Li, and Raghuwanshi Kailash: Evaluation of Different Electronic Product Code Discovery Service Models. Advances in Internet of Things (AIT), 2(2), 37-46, Scientific Research Publishing, 2012.

Posters

Chandrasekhar Bhagavatula, Kevin Iacovino, **Su Mon Kywe**, Lorrie Faith Cranor, and Blase Ur: Usability Analysis of Biometric Authentication Systems on Mobile Phones. The 10th Symposium On Usable Privacy and Security (SOUPS 2014), California, USA, July 9-11, 2014.

Chapter 1

Introduction

Mobile devices play a critical role in modern life and there are billions of mobile users around the world. It is a great responsibility of platform providers to ensure the security and privacy of mobile users. In this dissertation, we study the security of application layer and framework layer of mobile systems. First, platform providers incorporate several security mechanisms, such as application sandbox and permission access control, to the framework layer of mobile devices. Sandbox mechanism isolates the code execution and data storage of applications on mobile devices to minimize the damage potentially caused by malicious applications. At the same time, access control mechanisms allow applications with appropriate permissions to access important resources on mobile devices. Second, in the application layer, platform providers inspect the applications when they are uploaded to the application stores. They consequently remove the malicious applications once they are detected.

There are several drawbacks in these mechanisms:

- Currently, there is no standard way of vulnerability analysis on both Android and iOS frameworks. The lack of framework-specific vulnerability analysis tools makes the security testing difficult potentially leading to various vulnerabilities. On the other hand, ensuring security of mobile frameworks is not trivial due to a wide variety of API types and vast presence of APIs. The frameworks are also constantly modified by various platform providers in a very fast pace mobile environment.

- Security controls on mobile applications have been focusing on malwares, which only account for a small percentage of mobile applications in the markets. There has been negligence over bad applications, which are less aggressive than malwares, but still violate developers policies, such as intellectual property right violations.

In this dissertation, we take the first step towards systematic analysis of vulnerabilities in mobile frameworks and detect policy-violating applications. My first two work focus on vulnerability analysis of mobile frameworks and my last two work study the policy-violating applications. In my first work, we perform vulnerability analysis on iOS framework. In my second work, we perform vulnerability analysis on Android framework. In both work, we consider a third-party application as attacker. This attacker gains access to mobile resources by bypassing the security mechanisms of mobile framework. The results of these two work are proof-of-concept attacks that can be performed on mobile frameworks. In my third work, we perform empirical analysis on the policy-violating applications and create detection mechanisms for all applications violating Google Play policies. After that, in my fourth work, we focus on detecting camouflaged applications, which are also part of policy-violating applications.

1.1 Identifying Vulnerabilities in iOS Framework

iOS is Apples mobile operating system, which is used on iPhone, iPad and iPod touch. Any third-party applications developed for iOS devices are required to go through Apples application vetting process and appear on the official iTunes App Store upon approval. When an application is downloaded from the store and installed on an iOS device, it is given a limited set of privileges, which are enforced by iOS application sandbox. Although details of the vetting process and the sandbox are kept as black box by Apple, it was generally believed that these iOS security mechanisms are effective in defending against malwares.

In this work, we propose a generic attack vector that enables third-party applications to launch attacks on non-jailbroken iOS devices. They include attacks via dynamically loaded frameworks and attacks via private C functions. With these attack vectors, an attacker obtains access to both public and private APIs of iOS framework. Following this generic attack mechanism, we are able to construct multiple proof-of-concept attacks. They include cracking device PIN, blocking phone calls, taking snapshots without users awareness, sending SMS and emails and posting tweets to Twitter. Our applications embedded with the attack codes have passed Apples vetting process and work as intended on non-jailbroken devices. Our proof-of-concept attacks have shown that Apples vetting process and iOS sandbox have weaknesses which can be exploited by third-party applications.

We further provide corresponding mitigation strategies for both vetting and sandbox mechanisms, in order to defend against the proposed attack vector. We suggest using fuzzing tests and dynamic taint analysis during Apple's vetting process. We also believe that iOS sandbox can be improved by dynamic parameter inspection, privileged IPC verification, service delegation enhancement, and system notifiers for sensitive functionalities.

1.2 Identifying Vulnerabilities on Android Framework

Android requires third-party applications to request for permissions when they access critical mobile resources, such as users' personal information and system operations. For instance, only applications with `android.permission.CAMERA` permission are given access to phone cameras.

In this work, we present the attacks that can be launched without permissions. We group the Android APIs into three categories: system services, system applications, and dynamically register broadcasts. To identify all vulnerabilities, we perform inter-procedural call graph analysis on system services and discover all Android Interface Definition Language (AIDL) interfaces that are not protected by any permission checking or Linux ID checking mechanisms. We then carry out a component analysis on system applications so as to locate the exposed and unprotected broadcast receivers, activities and services. After that, we conduct an intra-procedural data-flow analysis to find out unprotected dynamically registered broadcasts from both system services and system applications. The result of our analysis is a systematic overview of unprotected Android APIs. These unprotected APIs provide a way of accessing resources without any permissions.

We then exploit selected unprotected APIs and launch a number of attacks on Android phones. In particular, we launch Java reflection attacks, broadcast injection attacks, broadcast hijacking attacks, malicious activity launch attacks, activity hijacking attacks, malicious service launch attacks, and service hijacking attacks. We discover that without requesting for any permissions, an attacker can access to device ID, phone service state, SIM card state, Wi-Fi and network information, as well as user setting information, such as airplane, location, NFC, USB and power modes of mobile devices. An attacker can also disturb Bluetooth discovery services, and block the incoming emails, calendar events, and Google documents. Moreover, an attacker can set volumes of devices and trigger alarm tones and ringtones that users personally set for their devices. An attacker can also launch camera, mail, music and phone applications even when the devices are locked.

We compare our research on two Android versions, and discover that as platform providers incorporate more APIs, the number of unprotected APIs increases and new attacks become possible. This is contrary to the common belief that the security of a new version should improve, since many security flaws in an old version are reported and fixed. We thus suggest platform providers to inspect Android frameworks systematically before releasing new versions.

1.3 Detecting Policy-Violating Applications

To ensure quality and trustworthiness of mobile apps, Google Play store imposes developer policies that cover various aspects, including intellectual property rights, spams and advertisements. Once an app is reported for exhibiting suspicious behaviors that violate app policies, it is removed from the store to protect users. Currently, Google Play store relies on mobile users' feedbacks to identify violations.

Our work takes the first step towards understanding these reported apps by performing empirical analysis on real-life app samples. We crawl 302 policy-violating Android apps, which are reported in the Reddit forum by mobile users and are later removed from the Google Play store. Our empirical analysis reveals that many violating behaviors have not been studied well by industry or research communities. We discover that 53% of the reported apps are either copying popular apps, such as Bejeweled Blitz, Candy Crush, Minecraft, Angry Bird, and Fruit Ninja or violating copy-rights or trademarks of brands, such as Adobe, Disney, Minion, Despicable Me, and Pikachu. Moreover, 49% of reported apps are violating ads policies by sending push notifications, adding homescreen icon and changing browser settings. Many apps also show malware-like behaviors, such as downloading malicious files to users' mobile phones, redirecting users to other apps on the market and accessing to users' PayPal account.

Based on our empirical analysis results, we extract 208 features for differentiating bad apps from benign apps. Our features cover use of brand names and other keywords, third-party libraries, network activities, meta data, permissions, and suspicious API calls originated from third-party libraries. The first three groups of features are derived from empirical analysis of our reported app samples, while the last three groups of features are based on their bad behaviors. We apply 10 machine learning classifiers on the extracted features to detect reported bad apps. Our experiment result shows that we can detect them with 86.80% true positive rate and 13.6% false negative rate. Our work highlights the problem of policy-violating apps and suggests reconsidering the current strategy in maintaining good quality mobile app markets.

1.4 Detecting Camouflaged Applications

Application plagiarism or application cloning is an emerging threat in mobile application markets. It reduces profits of original developers and sometimes even harms the security and privacy of users. In this work, we introduce a new concept, called camouflaged applications, where external features of mobile applications, such as icons, screenshots, application names or descriptions, are copied.

We then propose a scalable detection framework, which can find these suspiciously similar camouflaged applications. To accomplish this, we apply text-based retrieval methods and content-based image retrieval methods in our framework. Our framework is implemented and tested with 30,625 Android applications from the official Google Play market. The experiment results show that even the official market is comprised of 477 potential camouflaged victims, which cover 1.56% of tested samples.

Our work highlights that these camouflaged applications not only expose potential security threats but also degrade qualities of mobile application markets. Our work also analyzes the behaviors of detected camouflaged applications and calculate the false alarm rates of the proposed framework.

1.5 Contributions and Impact

We summarize the contributions of this dissertation in the following:

- We perform different types of vulnerability analysis to uncover APIs that can be mis-used by malicious applications. Our analysis ranges from reverse engineering of iOS framework to call graph analysis and data flow analysis on Android framework. We use a third-party application as an attacker, which can bypass default security mechanisms of mobile frameworks. We are able to launch serious proof-of-concept attacks in both iOS and Android platforms. These attacks have been reported to respective platform providers, and are fixed in later versions of mobile frameworks.

- We take the first step towards understanding policy-violating applications by performing empirical analysis on real-life application samples. We also create several detection mechanisms based on machine learning and information retrieval algorithms. Our mechanisms are shown to be effective in detecting policy-violating applications.

Our work have great impact on mobile security industry affecting billions of mobile users worldwide. Our findings on iOS framework have been reported to Apple security team. Our team engaged in a conference call with them to assist them in understanding and patching vulnerabilities. Apple assigns Common Vulnerabilities and Exposures (CVE) number “CVE-2013-0957” to the reported vulnerabilities and publicly acknowledges our contributions [4]. The vulnerabilities are patched in new iOS versions. Moreover, our discovery has been reported by the news media including Straits Times [80] and Singapores Today newspapers [70]. Our findings on Android framework have been reported to Google and Google has publicly acknowledged our contribution in its Security Bulletin in March 2016 [35]. The reported vulnerabilities are also assigned with CVE number “CVE-2016-0831 ” and later are fixed on new Android version. Moreover, we obtained cash reward from the Android Security Rewards Program for reporting these vulnerabilities.

1.6 Organization of the Dissertation

The reminder of this dissertation is organized as follows: Chapter 2 investigates vulnerabilities of iOS framework and Chapter 3 studies vulnerabilities of Android framework. Chapter 4 conducts empirical analysis on policy-violating applications and discusses their detection algorithm. Chapter 5 proposes a method for detecting camouflaged applications. Chapter 6 summarizes the contributions of this dissertation and discuss future directions.

Chapter 2

Launching Generic Attacks on iOS with Approved Third-Party Applications

2.1 Introduction

Digital mobile devices, such as smartphones and tablets, have been increasingly used for personal and business purposes in recent years. iOS from Apple is one of the most popular mobile operating systems in terms of the number of users. By Jan 2013, 500 millions of iOS devices had been sold worldwide and Apples iTunes App Store contained over 800,000 iOS third-party applications, which had been downloaded for more than 40 billion times [45].

Third-party applications are pervasively installed on iOS devices as they provide various functions that significantly extend the usability of the mobile devices. On the other hand, these third-party applications pose potential threats to personal and business data stored on the devices. Thus, Apple adopts various security measures on its iOS platform to protect the device from malicious third-party applications. Among these security measures, Apples application vetting process and the iOS application sandbox are considered as the fundamental mechanisms that protect users from security and privacy exploits.

Each iOS third-party application is required to go through a vetting process before it is published on the official iTunes App Store, which is the only source of obtaining applications without jailbreaking an iOS device. Although details of the vetting process are kept secret, it is generally regarded as highly effective since no harmful malware on non-jailbroken devices has been reported on iTunes App Store [75] [29]. Only graywares, which stealthily collect sensitive user data, were found on iTunes Store. These graywares were immediately removed from the store upon discovery [81].

When an application is downloaded and installed on an iOS device, it is given a limited set of privileges [39], which are enforced by the application sandbox. With the sandbox restrictions, an application cannot access files and folders of other applications. In order to access the required user data or control system hardware (e.g. Bluetooth or WiFi), applications need to call respective iOS APIs which are hooked by the sandbox so that validations of these API invocations are performed dynamically. The sandbox mechanism serves as the last line of defense which restricts malicious applications from accessing privileged system services, abusing user data or exploiting resources of other applications.

Due to the closed-source nature of iOS platform, the implementation details of security mechanisms used by iOS (including vetting process and application sandbox) are not officially documented. As a result, to our best knowledge, there is no systematic security analysis conducted for iOS platform, which has been generally believed as one of the most secure commodity operating systems [57].

In this work, we make the first attempt in constructing generic attacks on iOS platform. Existing ad hoc attacks usually require root privilege [62] [63] [18] and thus work only on jailbroken iOS devices. In contrast, our attacks are intended to work on non-jailbroken iOS devices, which are protected by both vetting process and application sandbox. Thus, we propose an attack vector which include two attack stages: 1) In the first stage, malicious applications which are embedded with attack codes need to pass Apples vetting process in order to appear in the official iTunes App Store; 2) In the second stage, after users have downloaded these applications onto their iOS devices, the attack codes need to bypass the restriction of the iOS sandbox in order to perform malicious functionalities. We realize both attack stages by exploiting the weaknesses of the vetting process and the iOS sandbox. With the proposed generic attack vector, we implement seven proof-of-concept attacks, such as cracking device PIN and taking screenshots without users awareness, which impose serious threats to the security and privacy of iOS users. Most of our attacks implemented work on both iOS 5 and iOS 6. We implement multiple iOS applications and embed our attack codes into these applications, which are then submitted to the iTunes App Store. These applications with attack codes have passed the vetting process and all our attacks work effectively on non-jailbroken iOS devices[1]. Our proof-of-concept attacks and further validation experiments indicate that the current vetting process and iOS sandbox have vulnerabilities that can be exploited by malicious third-party applications to escalate their privileges and launch serious attacks on non-jailbroken iOS devices.

In order to defend against the proposed attacks, we further discuss several mitigation methods which could enhance both vetting process and iOS application sandbox. Some of these methods utilize existing iOS security features, thus can be conveniently implemented and deployed on the current iOS platform. We have notified Apple all of our findings and shared all our attack codes with Apples product security team. By the time the paper was accepted, Apple is still in the progress of addressing the security issues we have discovered.

In summary, this chapter makes the following contributions:

- We provide a generic attack vector which exploits the weaknesses of both vetting process and iOS application sandbox. The attack vector consists of two attack stages and can be used to construct serious attacks that work on

[1]Due to privacy concerns, we embedded secret triggers in our applications so that public users will not be affected by the attack codes in these applications.

non-jailbroken iOS devices.

- We implement seven proof-of-concept attacks with the attack vector proposed. We embed these attack codes into multiple applications we implemented and all the applications are able to pass the vetting process and appear on official iTunes Store.
- We suggest several mitigation methods to defend against our attacks. These methods include improvements on both the vetting process and the application sandbox, which can be deployed on the iOS platform conveniently.

The rest of the chapter is organized as follows. In Section 3.2, we define the background and our threat model. In Section 3.3, we describe generic attack vector, and in Section 3.4, we show a set of proof-of-concept attacks. In Section 2.4, we discuss mitigation strategies. In Section 3.6, we provide some discussions on our research. In Section 3.7, we summarize the related work. Finally, we conclude the chapter in Section 2.7.

2.2 Background and Threat Model

2.2.1 iOS Platform Overview

iOS platform follows a closed-source model, where source code of the underlying architecture and implementation details of its security mechanisms are not available to the public. Though it is debatable whether such obscurity provides better security, iOS has been generally believed as one of the most secure commodity operating systems [57]. Unlike other mobile platforms, third-party applications on iOS are given a more restricted set of privileges [39]. In addition, any third-party application developed for iOS must go through Apples application vetting process before it is published on the official iTunes App Store. While some users and developers favor to have such restrictions for better security, others prefer to have more controls over the device for additional functionalities, such as allowing to install pirated software and allowing applications to change the themes of the device. To attain such extended privileges, an iOS device needs to be jailbroken. Jailbreaking is a process of installing modified kernel patches which allow a user to have root access of the device so that any unsigned third-party applications can run on it. Although jailbreaking is legal [20], it violates Apples End User License Agreement and voids the warranties of the purchased devices. Jailbreaking is also known to expose to potential security attacks [62] [63].

Application Vetting Process. Without jailbreaking a device, the only way of installing a third-party application on iOS is via the official iTunes App Store. Any application that is submitted to iTunes Store needs to be reviewed by Apple before it is published on the store. This review process is known as Apples application vetting process. The vetting covers several aspects, including detection of malware, detection of copyright violations, and quality inspection of submitted applications. Although the vetting process is kept secret by Apple, it is generally regarded as highly effective as no harmful malware has been reported on iTunes Store [75] [29]. Only grayware (which stealthily collects user data) had been reported and was removed from the store upon reporting [29] [81].

Application Sandbox. iOS utilizes another security measure application sandbox to restrict privileges of third-party applications running on a device. The sandbox is implemented as a set of fine-grained access controls, enforced at the kernel level. Under the sandbox restrictions, an application cannot access files and folders of other applications. In order to access user data or control system hardware, applications also need to call respective Application Programming Interfaces (APIs) provided on iOS. These APIs are hooked by the sandbox so that validations of API invocations can be performed dynamically. The sandbox serves as the last line of security defense which limits malicious applications from accessing system services or exploiting resources of other applications.

iOS Frameworks and APIs. To facilitate development of third-party applications, a collection of frameworks are provided in Cocoa Touch [11], which include both public frameworks and private frameworks. Public frameworks are application libraries officially provided to third-party developers while private frameworks are intended only for Apples internal developers. Each framework provides a set of APIs with which applications can access required system resources and services. Similar to frameworks, APIs can also be categorized into public APIs and private APIs.

Public APIs allow third-party applications to access a limited set of user information and control hardware of iOS devices, such as camera, Bluetooth and WiFi. In contrast, private APIs are the APIs that are meant to be used by Apples internal developers. Private APIs may exist in both public and private frameworks. Though not officially documented, private APIs include various functions which could be used by a third-party application to escalate its restricted privileges. Thus, Apple explicitly forbids third-party developers from using private APIs and rejects applications once the use of private APIs is detected. On the other hand, private APIs can still be used by applications that are designed to run on jailbroken devices. Such applications are available through Cydia [32], which is an unofficial application market built for jailbroken iOS devices.

2.2.2 Threat Model

In this work, we are interested in finding out the possible attacks which can be performed by third-party applications on non-jailbroken iOS devices, as illusrated in Figure 2.1. The success of such attacks depends on two major factors: 1) whether the corresponding malicious applications can pass Apples vetting process and appear in the official iTunes App Store; and 2) whether malicious function calls can bypass the restriction of the iOS sandbox. We embed all our proof-of-concept attack codes in the applications we develop, which have passed Apples vetting process and have been digitally signed by Apple. Thus, our attacks embedded in these applica-

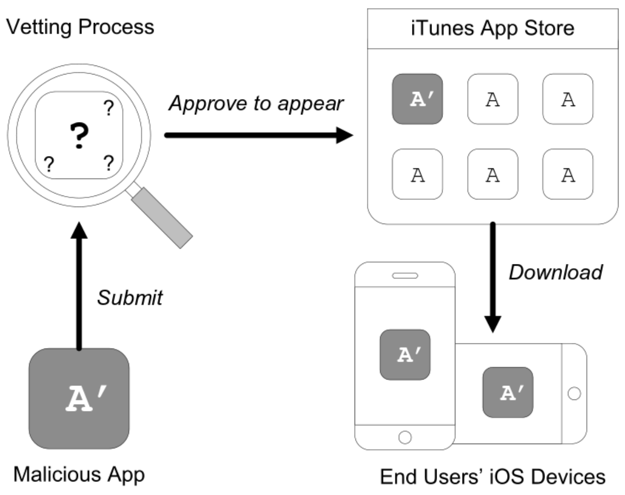


Figure 2.1: Threat Model

tions are able to work on both jailbroken and non-jailbroken iOS devices.

2.3 Generic Attack Vector

As introduced in Section 2, iOS private APIs exist in both private frameworks and part of public frameworks. When used by third-party applications, private APIs may provide additional privileges to the applications and thus are explicitly forbidden by the vetting process. We choose to utilize private APIs to construct our attacks which perform various malicious functionalities. In this section, we first present two ways of dynamically invoking private APIs which enable the malicious applications to pass the vetting process without being detected. Such dynamic loading mechanisms guarantee the success of the first stage in the proposed attack vector. For the second attack stage, in order to identify useful private APIs that are not restricted by iOS application sandbox, we manually analyze and test each iOS framework. Utilizing the useful private APIs we identified, we manage to implement multiple serious attacks that cover a wide range of privileged functionalities. These attacks can be embedded in any third-party applications, and they work effectively on non-jailbroken iOS devices. Although our attack vector includes two stages, these two stages are not isolated - what private API needs to be utilized decides the way of its dynamic invocation. Thus, in the following, we first use SMS-sending and PIN-cracking attacks as two examples to explain the underlying mechanisms of the entire attack vector. We then introduce other attacks we implemented utilizing the same attack vector and discuss the implications of these attacks.

2.3.1 Attacks via Dynamically Loaded Frameworks

When implementing a third-party iOS application that uses private APIs, the normal process is to link the corresponding framework statically (in the applications Xcode [23] project), and import the framework headers in the applications source code. For example, if a developer wants to send SMS programmatically in his application, `CoreTelephony.framework` needs to be linked, and `CTMessage-Center.h` needs to be imported in the application code. After preparing those preconditions, the SMS-sending private API can then be called as follows:

```
[[CTMessageCenter sharedMessageCenter]
sendSMSWithText:@"A testing SMS"
serviceCenter:nil
toAddress:@"+19876543210"];
```

In the above code, the static method `sharedMessageCenter` returns an instance of `CTMessageCenter` class, and then invokes the private API call `sendSMSWithText:serviceCenter:toAddress:`, which performs the SMS-sending functionality on iOS 5. Third-party application can utilize this method to send premium-rate SMS, and the sent SMS will not even appear in the SMS outbox (more precisely, it does not appear in the default iOS Message application[2]). Thus, a user would be totally unaware of such malicious behavior until the user receives his next phone bill.

However, this standard way of invoking private APIs can be easily detected by

[2]Another way of sending SMS programmatically on iOS 5 is to utilize `MFMessageComposeViewController`. However, this method is easy to be noticed as the SMS sent would appear in the default Message application.

the vetting process, even though only the executable binary of the compiled application is submitted for vetting. One way of detecting this API call is to simply use string matching (e.g., grep) on the binary, as the name of the function call appears in the binarys `objc_methnamesegment` (and also other segments). Moreover, the framework name and class name also appear in the binary as imported symbols. In this example SMS-sending code, although `CoreTelephony` is a public framework, `CTMessageCenter.h` is a private header (i.e., `CTMessageCenter` is a private class); thus, importing it in the source code can be detected by performing static analysis on the applications binary file. In order to pass Apples vetting process, the application cannot link the framework statically.

To avoid being detected, the framework has to be loaded dynamically and the required classes and methods need to be located dynamically. In our attacks, we utilize Objective-C runtime classes and methods to achieve this goal. The example SMS attack code that illustrates the dynamic loading mechanism is given as follows:

```
NSBundle *b = [NSBundle bundleWithPath:@"/System/Library
/Frameworks/CoreTelephony.framework"];
[b load];
Class c = NSClassFromString(@"CTMessageCenter");
id mc = [c performSelector:NSSelectorFromString(@"sharedMessage
Center")];
// call "sendSMSWithText:serviceCenter:toAddress:" dynamically
by utilizing NSInvocation
```

In the above code, the first two lines are used to load the CoreTelephony framework dynamically, without linking this framework in the applications source code. The path of this library is fixed on every iOS device, which is under the `/System/Library/Frameworks/` folder. Note that not only public frameworks can be loaded dynamically, private frameworks (which is under `/System/Library/Private-Frameworks/`) can also be loaded dynamically using the same method. According to our experiments, Apples sandbox does not check the parameter of `[NSBundle load]` to forbid accessing these frameworks under `/System/Library` folder.

`NSClassFromString` at the third line is a function which can locate the corresponding class in memory by passing it the class name, which is similar to the "`Class.forName()`" method in Java reflection. At the fourth line, the `sharedMessage-Center` method is called via "`performSelector:`". At last, in order to call a method with more than 2 parameters (which is "`sendSMSWithText:serviceCenter:toAddress:`"in this case), the `NSInvocation` class is utilized.

Although the above code dynamically invokes the private API call, it may need certain obfuscation in order to avoid the detection from static analysis during the vetting process[3]. The last step of generating the actual attack code is to obfuscate all the strings appearing in the above example code. There are various ways of obfuscating strings in the source code. One simple technique is to create a constant string which includes all 52 letters (both upper and lower cases), 10 digits and common symbols. Then all the strings appeared in the above code can be generated dynamically at runtime by selecting corresponding positions from this constant string. Some of our applications utilize this method to obfuscate strings in the attack codes, and some others adopt a complex obfuscation mechanism, which involves bitwise operations and certain memory stack operations that are more difficult to be detected.

2.3.2 Attacks via Private C Functions

Information about private Objective-C classes and methods in the Cocoa Touch frameworks can be obtained from the iOS runtime headers [65], which are generated using runtime introspection tool such as RuntimeBrowser [67]. An example of directly utilizing these Objective-C private APIs has been introduced in the previous subsection. However, Objective-C private classes and methods are not the only

[3]Actually according to our experiments, obfuscation may not be necessary, as the vetting process does not seem to check all text segments in the binary. In our experiments, we have tried to embed this SMS-sending code in one application which does not utilize obfuscation, and the application passed the vetting process.

private APIs we are able to use in third-party applications.

When we reverse engineer the binary files of each framework, we find that there are a number of C functions in these frameworks that can be invoked by our application, which do not appear in the iOS runtime headers [65] and cannot be found with RuntimeBrowser [67]. In order to invoke these C functions, we need to dynamically load the framework binary and locate the function at runtime. The following code segment is part of our PIN-cracking code, which illustrates how we realize the dynamic invocation for private C functions.

```
void *b = dlopen("/System/Library/PrivateFrameworks
/MobileKeyBag.framework/MobileKeyBag", 1);
int (*f)(id, id, id) = dlsym(b, "MKBKeyBagChangeSystemSecret");
...
int r = f(oldpwd, newpwd, pubdict);
...
```

In the above code segment, we use `dlopen()` to load the binary file of the private framework `MobileKeyBag`, which returns an opaque handle for this dynamic library. Utilizing this handle and `dlsym()`, we are then able to locate the address where the given symbol `MKBKeyBagChangeSystemSecret` is loaded into memory. This address is then casted into a function pointer so that it can be directly invoked later on in our attack code. Although the above code segment may look simple, it is actually not easy to identify which C functions we should invoke to serve for our attack purpose, especially when only framework binary is given. Even after the C functions are identified and located, it takes further tedious work to figure out the correct parameter types and values to pass to the C functions. And in many cases, even all parameters are correct, these functions may be restricted by iOS sandbox and thus will not function correctly within third-party applications. To speed up the manual reverse engineering process when analyzing the given framework binaries, we build our own static analysis tool (which is based on IDA Pro.) to disassemble the framework binary and obtain assembly instructions that are relatively easy to read.

By manually analyzing the private framework ManagedConfiguration, we find out that the changePasscodeFrom:to:outError: method of MCPasscodeManager is used to reset the password of the iOS device. However, we are not able to directly invoke this Objective-C method because the device needs to be "unlocked" first with current device password (possibly due to sandbox restrictions). Thus, we need to find a way of bypassing such restriction. Digging into the assembly code of the changePasscodeFrom:to:outError: method, we find out that it eventually invokes the MKBKeyBagChangeSystemSecret C function in MobileKeyBag to reset the password, which is allowed to be directly invoked under the sandbox restrictions. Further analysis and experiments are then conducted to figure out the correct parameters used to invoke MKBKeyBagChangeSystemSecret.

Our analysis reveals that the MKBKeyBagChangeSystemSecret function accepts three parameters, all of which have the type of (NSData*). The first parameter is the data of the old password, which can be converted from password string. The second parameter is the data of the new password. The third parameter, however, is an NSDictionary containing the keyboard type of the current password, which must be converted into NSData with [NSPropertyListSerialization dataFromPropertyList:format:errorDescription:]. One simple way of obtaining this NS-Dictionary data is to utilize the private framework ManagedConfiguration. However, in our attack code, to minimize the number of frameworks loaded, we utilize another private C function MKBKeyBagCopySytemSecretBlob[4] in MobileKeyBag to obtain this NSDictionary, which is then passed to MKBKeyBagChangeSystemSecret as the third parameter.

[4]Note that it is not a spelling error in this MKBKeyBagCopySytemSecretBlob function. The key word "System" in this function name is spelled as "Sytem" by Apples programmers. This detail further shows that in this attack, we utilize a function which Apple programmers may not expect to be used by third-party applications.

After this `MKBKeyBagChangeSystemSecret` function is successfully invoked, the rest of the attack code is straight forward we simply use brute force to crack the password. 4-digit PIN has been widely used to lock iOS devices and has a password space of 10^4. When using our application to crack a device PIN on iPhone 5, it takes 18.2 minutes on the average (of 16 trials on two iPhone 5 devices) to check the whole PIN space (10^4). This gives an average speed of 9.2 PINs per second. To further speed up the cracking, we build a PIN dictionary so that common PINs are checked first. If the given PIN is in birthday format (mmdd/ddmm), it takes about 40 seconds to crack the PIN on average. Note that since our PIN-cracking attack uses the low level C functions, it will not trigger the "wrong password" event on the iOS device which is implemented at higher level (Objective-C functions) in the framework code. Thus, there is no limit on the number of attempts for our brute force attacks when cracking the device PIN. It is the same procedure to crack 4-digit PIN and complex password using our method, but the latter will take much longer time than PIN due to its large password space.

2.3.3 Other Implemented Attacks and Implications

The SMS-sending attack and the PIN-cracking attack introduced above explain how the entire attack vector is constructed. The former uses private Objective-C functions (Section 3.1), while the latter uses private C functions (Section 3.2). With the same dynamic invocation mechanisms which are able to bypass the vetting process, other attacks can also be implemented, as long as we can identify sensitive private APIs that are overlooked by the iOS sandbox.

We manually analyze the 180+ public and private iOS frameworks and manage to identify seven sets of sensitive APIs that are not restricted by iOS sandbox. Utilizing these APIs and the dynamic invocation mechanisms, we implement seven attacks, which are listed in Table 2.1. The corresponding frameworks and key APIs utilized are listed in Table 2 in the appendix. We embed our attack codes in multi-

Attack Name	**Description**	**iOS 5**	**iOS 6**	**iPhone**	**iPad**
PIN-cracking	Crack and retrieve the PIN of the device	✓	✓	✓	✓
Call-blocking	Block all incoming calls or the calls from specified numbers	✓	✓	✓	-
Snapshot-taking	Continuously take snapshots for current screen (even the app is at background)	✓	✓	-	✓
Secret-filming	Open camera secretly and take photos or videos without the users awareness	✓	✓	✓	✓
Tweet-posting	Post tweets on Twitter without users interaction	✓	✓	✓	✓
SMS-sending	Send SMS to specified numbers without the users awareness	✓	-	✓	-
Email-sending	Send emails using users system email accounts without the users awareness	✓	-	✓	✓

Table 2.1: The Seven Attacks Implemented and their Applicability

ple applications we develop, and all those applications have passed Apples vetting process and appeared in the official iTunes App Store.

Most of the attacks in Table 2.1 work on both iOS 5 and iOS 6 (which is the default iOS version on iPhone 5). The call-blocking and SMS-sending attacks do not work on iPad, simply because iPad does not have corresponding functionalities since it is not a phone device. The secret-filming attack can be implemented purely with iOS public APIs. The last two attacks (SMS-sending and email-sending) currently only work on iOS 5, but not iOS 6. The APIs of sending SMS and emails on iOS 6 have been substantially changed to prevent such attacks (which will be further analyzed in Section 4).

The severity of most of our attacks would be significantly increased when the attack code is embedded in an application that can keep running at the background. Take the snapshot attack as an example. By calling the private API `[UIWindow createScreenIOSurface]`, an application can capture the current screen con-

Attacks	Frameworks	Classes	Functions
PIN-cracking	MobileKeyBag	-	MKBKeyBagChangeSystemSecret MKBKeyBagCopySytemSecret-Blob
Call-blocking	CoreTelephony	-	CTTelephonyCenterGetDefault CTTelephonyCenterAddObserver CTCallCopyAddress CTCallDisconnect
Snapshot-taking	UIKit	UIWindow UIImage	createScreenIOSurface initWithIOSurface:
Secret-filming	AVFoundation CoreMedia CoreVideo	AVCaptureDevice AVCapture-DeviceInput AVCaptureV-ideoDataOutput AVCaptureSes-sion	devices deviceInputWithDevice:error: setSampleBufferDelegate:queue: startRunning
Tweet-posting	Twitter	TWTweetCompose ViewController	setCompletionHandler: setInitialText: send:
SMS-sending	CoreTelephony	CTMessageCenter	sharedMessageCenter sendSMSWith-Text:serviceCenter:toAddress:
Email-sending	Message AppSupport	MailAccount CPDistribut-edMessaging-Center	defaultMailAccountForDelivery uniqueId centerNamed: sendMessageAndReceiveReply-Name:userInfo:error:

The symbol of “-” in the Class field indicates that the corresponding attack does not utilize any Objective-C classes, but only utilizes private C functions.

Table 2.2: The Frameworks and Key APIs Utilized for the Seven Attacks Implemented

tent of the device. When continuously running at the background, this application can take snapshots of the device periodically, and send these snapshots back to the developers server for further analysis[5]. Such snapshot-taking attack may reveal users email content, photos and even bank account information, thus it should be avoided on any mobile devices.

Similar to the snapshot-taking attack, the call-blocking and PIN-cracking attacks also become more serious when they are used in an application that can continuously run at the background, which have been verified in our experiments. However, the secret-filming attack does not work when in background. The current implementation of the iOS camera service requires that an application utilizing this service be not in the background status. Nevertheless, even if the secret-filming attack works only when the application is in the foreground, it is still a serious threat to user privacy. Considering that when a user is playing a game on the iOS device, and the game secretly opens the cameras and takes photos periodically without the users notice. In our experiments, we have verified that both front and back cameras can be used, and the sound can be muted when taking videos or photos programmatically in our applications.

We emphasize that all these attacks are implemented with secret triggers in the applications that are submitted to iTunes Store. The attacks are only launched on our testing devices after certain sequences of secret buttons have been pressed in the applications. However, note that in the application codes, such triggers are just "if-else" statements. Thus, if the trigger conditions were replaced with an "if-true" condition, these attacks could be launched on any user device with such applications. Therefore, the secret triggers used in our proof-of-concept applications do not affect the conclusions drawn from our experiments.

Besides the seven attacks we have implemented, our attack vector can be used to construct other attacks as long as there are security sensitive functions on iOS that

[5]The snapshot attack code is embedded into one of our applications which can keep running at background utilizing audio playing feature. This application also passed Apples vetting process and it sends out snapshots every 5 seconds once triggered.

are not restricted by iOS sandbox. As each iOS version will include new functionalities to the platform, each iOS update may introduce new attacks from malicious third-party applications based on our attack vector.

2.4 Attack Mitigation

Our proof-of-concept attacks have shown that Apples current vetting and sandbox mechanisms have weaknesses which can be exploited by third-party applications to escalate their privileges and perform serious attacks on iOS users. In this section, we first suggest improvements on the vetting process to mitigate the security threats caused by dynamic invocations. We then propose enhancements on the iOS sandbox to further defend against our attacks utilizing private APIs.

2.4.1 Improving Application Vetting Process

Static analysis can be used to determine all the API calls which are not invoked with reflection (i.e., dynamic invocations), and it can provide the list of frameworks that are statically linked in the application. Thus, an automated static analysis is able to detect the standard way of invoking private APIs, as what is probably being used by Apple in its current vetting process. In addition, we suggest to improve the existing static analysis to detect suspicious applications based on certain code signatures. For example, one suspicious code signature could be applications containing any `dlopen()` or `[NSBundle load]` invocations whose parameters are not constant strings (which match the cases of our attacks). However, as the static analysis alone is not sufficient to determine whether a suspicious application is indeed a malware or not, manual examination and dynamic analysis should be utilized to examine such suspicious applications.

In many cases, manual examination may not be able to find malicious behaviors of the examined applications, because the malicious functions may not be preformed for every execution. Instead, they can be designed in the way that such functions are only triggered when certain conditions have been satisfied. Examples of such conditions include time triggers or button triggers (as what have been used in our applications). When a malicious application uses such trigger strategy, the manual inspection may not find any suspicious behaviors during the vetting process. Such malicious applications can only be detected by utilizing fuzz testing [45] (or in the extreme case, using symbolic execution [17]), where different inputs are used to satisfy every condition of the application code. Furthermore, in order to determine whether sensitive user data are transferred out of the device, dynamic taint analysis [48] is an effective approach to serve this purpose. However, since it is expensive to apply fuzz testing and dynamic taint analysis on every application, the vetting process may choose to run such examinations only on selected suspicious applications.

2.4.2 Enhancement on iOS Sandbox

Dynamic Parameter Inspection. From the perspective of iOS sandbox, a straightforward defense to our attacks that utilize the dynamic loading functions (such as `[NSBundle load]` and `dlopen()`) is to forbid third-party applications to invoke these functions. However, it is not practical to completely forbid the invocation of dynamic loading functions, since frameworks, libraries and many other resources need to be dynamically loaded for benign purposes at runtime. Even Apples official code, including both framework code and application code (which is automatically generated by Xcode), utilizes dynamic loading functions extensively to load resources at runtime. On the other hand, since sensitive APIs can be hooked by utilizing the application sandbox, the parameters of these APIs can be checked at runtime. Thus, it is useful if Apples sandbox is modified in the way that the parameter values passed to dynamic loading functions are examined, and accessing files under a specific folder is forbidden.

One way of implementing this approach is to forbid the third-party applications to dynamically load any frameworks under "`/System/Library/`" folder.However, a sophisticated attacker may be able to completely reverse engineer a given framework binary, locate all the code regions in the binary that are needed for launching his attack, and then copy only the needed code regions from the binary and insert into his application code. In this way, he does not need to dynamically load framework binaries in his malicious applications. Therefore, this parameter-inspection approach is not able to completely defend against the proposed attacks, though it can increase the complexity for the adversary to construct these attacks.

Privileged IPC Verification. Another technique of enhancing the sandbox is to dynamically check the privilege of the identity which makes sensitive API calls. For example, a third-party application should not have the privilege to invoke `MKBKeyBagChangeSystemSecret` API,which is used in our PIN-cracking attack. Such private APIs should only be invoked by processes or services with the system privilege. However, directly restricting the access to private APIs may not effectively prevent the attacks. By analyzing the implementation of several private APIs (in assembly code), we find that the private APIs eventually use inter-process communication (IPC) methods, which communicate with the system service process, to complete the functionalities of the private APIs. For example, `MKBKeyBagChangeSystemSecret` API uses `perform_command()` method to communicate with the system service (with `service bundle id = ‘‘com.apple.mobile.keybagd’’`). This means that instead of invoking private APIs, an application can also use such IPC method to directly send command to the system service process to perform the same functionality.

In order to defend against such attacks, for each privileged system service, the recipient of the command (which is the service process itself) needs to check the sender of the command to verify whether the sender has the valid privilege to make such IPC. To enable this IPC verification, the system service process needs to maintain a list of privileged IPC commands which are checked dynamically when an IPC is received. Compared to the parameter-inspection approach, privileged IPC verification provides better defense against the PIN-cracking, call-blocking and snapshot-taking attacks as the corresponding privileged functionalities should not be used by any third-party applications. However, this approach alone is not sufficient to mitigate the other four attacks listed in Table 1. For these four attacks, the corresponding functionalities should be provided to applications due to usability reasons, but at the same time, it needs to be ensured that user interactions are involved when these functionalities are performed.

Service Delegation Enhancement. On iOS 6, Apple starts using the XPC Service, which allows processes to communicate with each other asynchronously so that it can be used for privilege separation. Originally on iOS 5, the SMS and email APIs are implemented as "View Controller" classes that are created and used within a third-party application process. Therefore, applications can manipulate these view controller classes to send out SMSes and emails programmatically without users interaction. However, on iOS 6, the SMS and email functionalities are now delegated to another system process utilizing XPC Service, which is completely out of the process space of third-party applications. Thus, a third-party application on iOS 6 is no longer able to send SMSes or emails programmatically without users interaction.

Although currently iOS 6 has not implemented the service delegation mechanism for the Twitter service, the tweet-posting attack can be prevented using this mechanism, as it follows exactly the same service model as SMS and email. The secret-filming attack, however, cannot be easily mitigated using such service delegation. Instead of using a unified user interface, iOS enables third-party applications to create their own customized user interfaces for taking photos or videos. If the same service delegation mechanism is applied, then the camera interface will be identical across different applications as it is provided by system service. Thus, more precisely, service delegation is able to defend against camera device abuse, but its implementation may greatly impact user experience.

System Notifiers for Sensitive Functionalities. In order to mitigate the threat of secret filming, while preserving the functionality and flexibility of using camera in third-party applications on iOS, one possible solution is to add a half-transparent system notifier on the screen (e.g., at the upper-right corner), whenever the camera device is being used. This notifier can be shown using the XPC mechanism so that the notifier is handled by a system daemon process, which is outside of the control of third-party applications. In this way, whenever the camera is being used (either taking photos or taking videos), the system notifier is shown on the screen to alert the user.

By enhancing the current iOS platform with the 1) privileged IPC verification, 2) comprehensive service delegation, and 3) extended system notifiers, it will be able to defend against all the seven attacks we construct. Note that since iOS is a close-source platform, it is extremely difficult (if not impossible) for us to implement these mitigation methods we proposed, and thus it is one of the limitations in our work. However, we have shared all our mitigation suggestions with Apple so that Apples product security team may choose some of these methods to fix the sandbox. From the partial knowledge that is revealed by our attacks and the mitigation analysis, it may be inferred that the current iOS sandbox implementation is quite complex and its privilege check is not complete. Due to its complexity and also its trade-off nature against usability, it may not be easy to completely fix the iOS sandbox to prevent future attacks.

2.5 Discussions

On the current iOS platform, when an application plays an audio file (e.g.,.mp3), normally a music-playing notifier (i.e. the ▶ symbol) is shown in the status bar on top of the screen. However, this only happens when the application is implemented following the standard programming rules, which require the application code to call `[[UIApplication sharedApplication] beginReceivingRemote-ControlEvents]`. This API call registers the application in the system service so as to receive remote events, such as when a user presses the control buttons on earphone. In the background running application we implement, however, this API is not invoked and our application simply calls the basic audio playing APIs to play a silent music in an infinite loop. As a result, no notifier is shown on the status bar when our application is running at the background, thus the iOS user may be totally unaware of the existence of this security threat. In addition to playing audio, there are other means of enabling background running, such as VOIP and tracking locations. Thus, besides the system notifier for the camera functionality (Section 4.2), we suggest to add another system notifier specifically designed to indicate that an application is running at the background. Upon seeing this notifier, a user can force close any background applications that are not being used. This will not only enhance security but also save device battery.

The PIN-cracking attack code introduced in Section 3.2 not only can be used to steal device PIN and send it to an external server, but can also be used to reset the current PIN to another value so that the legitimate user is not able to unlock the device. In iOS settings, there is an option to "erase all data on this device after 10 failed passcode attempts". If this option is enabled on a device and our PIN-cracking code resets the PIN, it could make a user panic if he is unable to unlock the device after several trials of inputting his original password. Again note that our PIN-cracking attack itself will not trigger the "wrong password" event on the iOS device and thus, there is no limit on the number of brute forcing trials for our attack code when cracking the device PIN.

With the attack codes we shared with Apples product security team, the PIN-cracking vulnerability has been fixed in the newly released iOS 6.1 (January 2013). However, other security issues we discovered are still in the process of being addressed. Note that the conclusions about the vetting process and sandbox given in this work are inferences based on observations from our experiments, as the details of the vetting process and sandbox are kept as blackbox by Apple. The ground truth may become available to the public when Apple decides to turn major components of iOS into open source in the future, as what has been done for Mac OS X [5].

2.6 Related Work

Spyphone [66] is a prototype application, developed for iOS 3.1.2, which illustrates that a wide list of user data can be accessed on iOS by third-party applications. However, Spyphone does not use any private APIs it only invokes public APIs and reads public files to access user data in order to enable itself to appear in iTunes Store, which is completely different from our malicious applications implemented. In addition, the security enforcement of iOS has been significantly improved since then so that a large portion of user data that can be accessed by Spyphone on iOS 3 is forbidden to access since iOS 5.

Malwares, such as iKee [62] and Dutch 5 ransom [63] worms, have been found on iOS. However, these worms only work on jailbroken iOS devices where an SSH server is installed with the default root password unchanged. Other iOS malwares known to the public, such as iSAM created by Damopoulos et al. [18] (which focuses more on malware propagation methods), also exploit vulnerabilities exist only on jailbroken iOS devices, which are different from our work.

Felt et al. [29] conduct a survey on the modern mobile malware in the wild, which encompasses all known iOS, Symbian, and Android malwares that spread between January 2009 and June 2011. They find that (i) all the 4 iOS malwares they identified work only on jailbroken iOS devices, and none were listed in the iTunes App Store; and (ii) only graywares are found on iTunes App Store which are then removed by Apple. These findings are confirmed by Egele et al. [24], in which they develop a static analysis tool, PiOS, to detect privacy leakages in iOS applications. They perform static analysis on more than one thousand third-party iOS applications and find out that only a few applications are graywares which stealthily access user data without users awareness.

Extensive researches have been conducted on the other popular mobile platform Android. Privilege escalation attacks on Android are proposed by [31], and the defense mechanisms for such attacks are introduced by Bugiel et al. [105]. Enck et al. [26] performs static analysis of Android applications using the decompiler they developed. Dynamic taint analysis on third-party Android applications is performed by TaintDroid [25]. Comprehensive surveys on mobile security are provided by Becher et al. [10] and Egners et al. [2].

The closest work to our research is the work by Miller [77]. By exploiting the security flaw he found, he managed to get iOS devices to run unsigned codes which are dynamically downloaded by his proof-of-concept malicious application. Millers attack mechanism provides an alternative for the first stage of our proposed attack vector. However, Apple has removed his application from the iTunes App Store and released a fix for the security flaw. Thus, our dynamic invocation used in the first stage, to our best knowledge, is the only way of bypassing the vetting process. Although our mechanism is not complex, it is a very effective way of allowing malicious applications appear in the official application store. Furthermore, by performing sophisticated analysis on all existing iOS frameworks, we identify seven sets of sensitive APIs which are not restricted by iOS sandbox and thus can be utilized by any malicious applications.

2.7 Conclusion

The original goal of this work is to answer a simple (but not easy) research question: is there a generic attack vector which enables third-party applications to launch attacks on non-jailbroken iOS devices? Two pre-conditions need to be satisfied in answering this question: (i) the third-party application has to pass the vetting process and appear on the official application store; and (ii) the corresponding attack codes must break through the restrictions of iOS sandbox in order to work on non-jailbroken iOS devices.

In this chapter, we constructed effective mechanisms which allow any third-party application to invoke private APIs without being detected by the vetting process. By utilizing such mechanisms and exploiting the vulnerabilities in the application sandbox, we implemented seven proof-of-concept attacks which can cause serious damages to iOS users. Finally, we suggested mitigation mechanisms to enhance the current vetting process and iOS sandbox. Our work fills the gap in the current mobile security literature where most research efforts are conducted on Android platform. We have shared all our findings with Apple's product security team. In January 2013, Apple released iOS 6.1 and fixed the PIN-cracking vulnerability we discovered in iOS 6.0, while other security issues presented in this chapter still remain unsolved.

Chapter 3

Attacking Android Smartphone Systems without Permissions

3.1 Introduction

Android adopts a permission system to protect users' security and privacy. To access resources that are out of application's sandbox, an application needs to request for permissions from users. In recent years, it has been reported that the Android permission system suffers from several flaws. For instance, unprivileged applications may leverage privileged applications to perform privileged tasks due to privilege escalation attacks [19] [12]. Several applications may collude to launch attacks with combined permissions from all of the applications [30] [76]. However, no rigorous study has been made on what potential attacks an application can launch on Android smartphone systems without requesting for any permissions. Therefore, in this work, we question the coverage of the current protection mechanisms and investigate to what extent critical resources are exposed to malicious applications via APIs without any protection mechanisms. There are two steps in our study. In the first step, we analyze unprotected Application Programming Interfaces (APIs), which allow third-party applications without any permissions to interact with mobile system resources, such as GPS and camera, or to access users' personal information. In the second step, we demonstrate a number of attacks that can be easily launched by leveraging on the unprotected APIs obtained in the first step.

To retrieve unprotected APIs from Android framework, we perform the following source-code static analysis: (1) inter-procedural call graph analysis on system services for the discovery of all Android Interface Definition Language (AIDL) interfaces that are not protected by any permission checking or Linux ID checking mechanisms, (2) component analysis on system applications for identifying the exposed and unprotected broadcast receivers, activities and services, and (3) intra-procedural data-flow analysis for locating unprotected dynamically registered broadcasts in both system services and system applications. We apply our analysis on Android Open Source Project (AOSP) versions 5.1.0_r1 and 4.4.0_r1. On AOSP version 5.1.0_r1, we identify 735 unprotected APIs in system services. In system applications, we discover 612 unprotected components, where 156 are unprotected broadcast receivers, 423 are unprotected activities and 33 are unprotected services. Moreover, we discover 206 unprotected dynamically registered broadcasts, where 50 exist in system services and 156 exist in system applications. It is alarming that a high number of unprotected APIs is discovered in different parts of Android frameworks. We also compare our analysis results on versions 5.1.0_r1 and 4.4.0_r1. We discover that the number of unprotected APIs increases on the newer version due to the newly added functionalities. This is contrary to the common belief that the security of a new version should improve, since many security flaws in an old version are reported and fixed.

After obtaining unprotected APIs, we create an adversary third-party application without any permissions, which launches Java reflection attacks, broadcast injection attacks, broadcast hijacking attacks, malicious activity launch attacks, activity hijacking attacks, malicious service launch attacks, and service hijacking attacks. We discover that on Android version 4.4.0_r1, an attacker can block the synchronization of emails, calendar events, browser bookmarks, browsing history, browser extension, Google documents, and Google notes. In addition, an attacker can send notifications, set car mode, set night mode, wake up the device at certain time, and set screen-off time. We reported our attacks on AOSP version 4.4.1_r1 to Google and some of the reported vulnerabilities are fixed on version 5.0.0_r1. Nonetheless, we still discovered more attacks on version 5.1.0_r1, which are also subsequently reported and fixed on version 5.1.1_r35 and version 6.0. This shows that while the platform providers make their effort in improving the security of Android framework, they need a powerful tool to win the “arms race”.

On version 5.1.0_r1, we discover that an attacker can obtain country, Wi-Fi information, subscriber information, tether state, airplane mode, NFC state, GSM/ CDMA strength, location mode, USB state, power state and security setting for lock screens. Moreover, some resources, such as device ID and SIM card state, which should be accessed by permission-granted applications only, are accidentally made available to all applications via unprotected APIs. An attacker can arbitrarily set the volumes of Android phones and play users’ incoming call ringtones, alarms, and notification sounds. An attacker can block Bluetooth discovery services, and launch camera, mail, music and phone system applications even when the targeted devices are locked. An attacker can also hijack various activities of system applications, including the interfaces for setting VPN (Virtual Private Network), Bluetooth and Wi-Fi, as well as the interfaces for adding device administrators and user accounts. These attacks show that the negligence in designing API-level permission enforcement causes various threats to users’ security and privacy. We suggest platform providers to systematically analyze unprotected APIs before releasing new versions, so that similar attacks are prevented in the future.

The rest of the chapter is organized as follows. In Section 3.2, we define our adversary model. In Section 3.3, we describe how we retrieve unprotected APIs. In Section 3.4, we show a set of proof-of-concept attacks on AOSP version 5.1.0_r1. In Section 3.5, we compare our results with AOSP version 5.1.0_r1. In Section 3.6, we provide some discussions on our research. In Section 3.7, we summarize the related work. Finally, we conclude the chapter in Section 4.7.

3.2 Adversary Model

Our adversary is a third-party application without any privileges or permissions, which launches malicious operations using unprotected Android APIs. We refer to it as "an attacker" in this work. We classify Android APIs into three categories: (1) normal APIs supported by *system services*, (2) loosely-coupled APIs supported by *system applications*, and (3) *dynamically registered broadcasts* in both system services and system applications. In the first category, API calls from applications are handled by system services, which provide main client-server interfaces between system-level processes and third-party application processes. For instance, to exercise a complete control over cameras, such as changing the zoom and flash light settings, an application may access `com.hardware.camera2` API, which in turn communicates with the system service, `android.hardware.ICameraService`. In the second category, loosely-coupled APIs are supported by system applications, which provide easy access to mobile phone functions. For instance, to take a photo or a video, an application may call the system application `Camera` using intents. Unlike normal APIs and loosely-coupled APIs, dynamically registered broadcasts in the third category are undocumented APIs. They are mainly used for internal communications among system services and system applications. All these types of APIs can be abused by an attacker when they are not properly protected.

3.2.1 System Services

Third-party applications access APIs of system services by calling the method `getSystemService(name)` of the `Context` class. The parameter `name` represents the name of the required system service. The returned object is then casted into the `Manager` class. For example, `AlarmManager` object can be retrieved by invoking the method with parameter "`alarm`". However, security checks performed inside the `Manager` class can be easily bypassed [27]. Moreover, APIs listed inside the `Manager` class are not complete; third-party applications can use Java reflection to invoke private APIs, which are marked with "@hide" annotations. Thus, we assume that an attacker may use Java reflection to interact with all unprotected APIs, including public and private APIs. Using Java reflection, an attacker invokes the `getService(name)` method inside the hidden `ServiceManager` class. Even though `ServiceManager` is a hidden class, it is unlikely to change, as the `android.jar` library relies on it to support normal APIs. The `getService(name)` method returns an `IBinder` object, which can be used to invoke any exposed methods inside the corresponding system services.

Listing 1 shows an example attack on AOSP version 4.4.4_r1. In this example, an attacker attempts to set the maximum screen-off time on mobile devices. There is no publicly available API for such function inside the `PowerManager` class, which is responsible for managing the power state of Android devices. However, `IPowerManager`, a hidden Stub class used by `PowerManager`, provides such method. To obtain an `IBinder` instance of `IPowerManager`, an attacker may first call `ServiceManager.getService("power")` method using Java reflection. After that, it invokes the `setMaximumScreenOffTimeoutFromDeviceAdmin()` method of `IPowerManager`, which in turn calls similar functions inside `PowerManagerService`. In this way, the attacker gains access to the system service running in privileged process. Since we set the screen-off time to 0, mobile users have no idea why their phone screens keep fading out. This unprotected API is originally intended for internal use, as `PowerManagerService` states in its comment, "Used by device administration to set the maximum screen off timeout. This method must only be called by the device administration policy manager." However, no security checking is performed inside `PowerManagerService`, which makes it vulnerable to malicious third-party applications.

Listing 1 An Example Attack using an Unprotected API from a System Service

```java
//Invoke ServiceManager.getService("power") method and obtain IBinder object of PowerManagerService

Class serviceManagerClass = Class.forName("android.os.ServiceManager");
Method getServiceMethod = serviceManagerClass.getDeclaredMethod("getService", String.class);
IBinder iBinder = (IBinder) getServiceMethod.invoke(null, "power");

//Get Stub object of IPowerManager by passing IBinder object to asInterface() method

Class stubClass = Class.forName("android.os.IPowerManager$Stub");
Method asInterfaceMethod = stubClass.getMethod("asInterface", new Class[]{IBinder.class});
Object IPowerManagerObj = asInterfaceMethod.invoke(null, iBinder);

//Invoke IPowerManager.setMaximumScreenOffTimeoutFromDeviceAdmin(0) method using Java reflection

Class IPowerManagerClass = Class.forName("android.os.IPowerManager");
Method setScreenOffTimeoutMethod = IPowerManagerClass.getDeclaredMethod("setMaximumScreenOffTimeoutFromDeviceAdmin", Integer.TYPE);
System.out.print(setScreenOffTimeoutMethod.invoke(IPowerManagerObj, 0));
```

3.2.2 System Applications

System applications provide loosely coupled APIs by exposing their components in the applications' `AndroidManifest.xml` files. In each XML file, `<application>` is the parent element, which contains some sub-elements for the application's components, such as `<service>`, `<activity>`, and `<receiver>`. Several tags and attributes are used to protect the components of system applications from other applications. They include `intent-filter`, `exported`, `permission` and `enabled`. Each exported and enabled component without any permission protection represents an unprotected API. We consider the following types of attacks.

- **Broadcast Theft:** An attacker may eavesdrop normal broadcast intents. This may result in certain user information being stolen by the attacker.

- **Malicious Broadcast Injection:** An attacker may send broadcasts and trigger broadcast receivers of system applications. This may result in unintended actions being performed by the attacker.
- **Activity Hijacking:** An attacker may launch its own activities when system activities are invoked. This may result in phishing attacks, where users mistake malicious interfaces as system interfaces.
- **Malicious Activity Launch:** An attacker may secretly launch activities from system applications if the activities are not well-protected. This may result in changing the state of certain system applications or tricking users.
- **Service Hijacking:** An attacker may intercept the intents sent to legitimate services of system applications. The attacker may provide false responses to the calling applications.
- **Malicious Service Launch:** An attacker may launch any unprotected services of system applications. The damage of this attack depends on the functionality of the unprotected services.

Our work is the first to consider these types of attacks on AOSP system applications and analyze them as a part of Android framework, although they have been applied on third-party applications and vendor-customized system applications by Chin et. al. [15] and Wu et al. [87] respectively. Note that we consider broadcast receivers of system applications or both broadcast theft and malicious broadcast injection attacks, so that we can discover as many attacks as possible on AOSP framework.

In addition to `intent-filter`, `exported`, `permission` and `enabled` tags and attributes used in `AndroidManifest.xml` files, the concept of *protected broadcasts* is used for limiting broadcast injection. Protected broadcasts are broadcasts that can only be sent by applications running in system-level processes. Protected broadcasts are defined in the `AndroidManifest.xml` file of AOSP root source code. For example, if the file includes the `<protected-broadcast android:name = "android.intent.action.PACKAGE_INSTALL"/>` tag, Android system allows no applications except system-level applications to send broadcasts with the action string, `android.intent.action.PACKAGE_INSTALL`. Thus, when launching broadcast injection attacks, we exclude these protected broadcasts from a list of our discovered broadcasts.

3.2.3 Dynamically Registered Broadcasts

Both system services and system applications may register broadcast receivers dynamically using APIs, such as `registerReceiver(BroadcastReceiver,IntentFilter)` or send broadcasts using APIs, such as `sendBroadcast(Intent)`. For simplicity, we refer to these types of broadcast receivers and broadcasts as "dynamically registered broadcasts" or simply "broadcasts" under related sections. Using them, an attacker may launch broadcast theft and broadcast injection attacks.

3.3 Retrieving Unprotected APIs

Retrieving unprotected APIs is not trivial due to a wide variety of API types, vast presence of APIs in Android framework and different protection mechanisms enforced. In this section, we apply three types of analysis for retrieving unprotected APIs: (1) call graph analysis on APIs provided by system services, (2) component analysis on APIs supported by system applications ,and (3) data flow analysis on dynamically registered broadcasts. The result of our analysis provides a broad overview of unprotected APIs in Android framework. Our analysis is first applied to AOSP version 5.1.0_r1 with API level 22 (Lollipop). We perform call graph analysis and data flow analysis based on Soot [50] version 2.5, which is an existing Java source code analysis tool. We also develop our own tool written in Python for scanning and identifying necessary source code files, and for analyzing components of system applications. We use an LG Nexus 5 for testing.

3.3.1 Call Graph Analysis on System Services

Using Android Debug Bridge (ADB) command, we discover 97 system services in Android 5.1.0_r1 version, where 11 of them are listed without any interface names. These 11 system services are designed to communicate with other system-level processes only. We identify all the unprotected APIs of system services using call graph analysis. A call graph is a directed graph, where each node represents a method and each edge indicates the invocation of one method to another. Our call graph analysis involves three steps: (1) finding all available APIs from a system service, (2) finding all security checking methods protecting the APIs, and (3) finding whether there exists at least one method call chain from an exposed API to any security checking method.

Step 1: Finding Source Methods - The *source* methods are the public methods of system services that are exposed via AIDL interfaces. We apply Soot to load a list of system service classes, and loop through all their public methods. In this way, we discover 1,751 APIs that are exposed to third-party applications.

Step 2: Finding Sink Methods - The *sink* methods are the methods that perform security checks. In this work, we consider permission and Linux ID checking mechanisms of system services. Some methods in Andriod framework are dedicated for permission checking [3], and we identify 35 of them, including 18 methods from the `ContextImpl` class, 2 methods from the `ActivityManager` class and 15 methods from the `PackageManagerService` class.

There is no specific method dedicated for Linux ID checking. System services normally perform the following steps for Linux ID checking. First, they obtain the UID or PID of the calling application or process using `getCallingPid()` and `getCallingUid()` methods from the `Binder` class. After that, they perform conditional check, such as "`callingUid != Process.SYSTEM_UID`", where `SYSTEM_UID` represents 1000. To locate ID checking methods, we first identify whether a method calls `getCallingPid()` and `getCallingUid()`. We then determine whether the returned variables are checked against any system-level Linux IDs in any `If` statements in the following source code. Note that the UIDs for system applications range from 0 to 9999. For instance, the UID for root user is 0, and the UID for telephony is 1001. The most commonly used UID is 1000, and it is used for running system server codes with certain privileges. As long as there exists at least one check against system-level IDs, we regard this method as a *sink* method.

Step 3: Building Call Graph - A context-insensitive inter-procedural call graph is built using Soot. The set of publicly accessible methods (i.e. *source* methods) are marked as entry points of the call graphs. After building the call graph, we loop through the method calls, and check if each *source* method ends up with any *sink* methods. We then exclude the methods with any security checking. In this way, we discover a list of methods that are not protected by any security mechanisms. Our analysis discovers 735 unprotected APIs, which count for 41.98% of total public APIs of system services. Our call graph analysis shows that a large number of Android APIs are unprotected and accessible by any third-party applications without any privileges.

3.3.2 Component Analysis on System Applications

We apply component analysis to retrieve unprotected components of system applications. There are altogether 69 system applications. We extract the component information from the `AndroidManifest.xml` files of system applications. We discover altogether 110 broadcast receiver components, 414 activity compo-

	Broadcast Receivers	**Activities**	**Services**
No of unprotected action strings	156	423	33
No of unique unprotected action strings	86	189	23
No of unprotected system applications	30	45	9

Table 3.1: Analysis Result of System Applications

nents and 140 service components from 69 system applications. A single system application component may have multiple `<intent-filter>` tags with multiple action strings. To discover unprotected action strings of system components, we first analyze if the system applications implement any application-level permissions. We then explore the components of system applications that satisfy the following conditions: (1) the components' attributes contain `intent-filter`, (2) `permission` is set to `none`, and (3) `exported` is set to `none` or `true`. In our analysis, we do not consider the `enabled` attribute, since it can be changed dynamically. Table 3.1 shows the result of our component analysis on system applications. Activities represent the most common type of unprotected action strings, followed by broadcast receivers and services.

Unlike other components, broadcasts can be further protected by Android system. Such broadcasts are called *protected broadcasts*, which are defined with `<protected - broadcast>` tag. Only system-level processes are allowed to send protected broadcasts. We obtain a list of protected broadcasts from the manifest file located under directory `frameworks/base/core/res/AndroidManifest.xml`. In total, we discover 225 protected broadcasts in the manifest file. Among the 86 broadcast action strings exposed from system applications, 34 of them are protected system broadcasts. Thus, an attacker may launch broadcast injection attacks with the remaining 52 broadcasts.

3.3.3 Data Flow Analysis on Dynamically Registered Broadcasts

Both system services and system applications may register and send broadcasts dynamically. We first retrieve the source code files of system services and system applications. After that, we apply data flow analysis to obtain the broadcast action strings from the source code files.

Identifying Dynamically Registered Broadcasts

From the AIDL interfaces obtained from the ADB command, we retrieve the Java files of system services. For instance, we find `AlarmManagerService` file from `IAlarmManager` AIDL interface. To achieve this, we scan the entire framework, and obtain Java files that (1) extend AIDL interfaces, (2) create new Stub classes with AIDL interface names or (3) implement AIDL interfaces and later extend them. These are the different ways by which system services implement their AIDL interfaces. In total, we discover 80 files of system services. The remaining services are only exposed via native codes, and thus excluded from our analysis. Note that our analysis does not include the classes called by the service class. A more complicated analysis will be required if we want to include them, since we will have to determine which methods are called from the service class. To obtain the source code of system applications, we scan the AOSP source code directories, read in every Java file, and look for package names of system applications. In total, we collect 1,392 source code files for 69 system applications. From the identified source code files of system services and system applications, we search for broadcast registering methods, such as `registerReceiver(BroadcastReceiver, IntentFilter)`, and broadcast sending methods, such as `sendBroadcast(Intent)`, of `Context` class. We apply data flow analysis on these methods so as to obtain the action strings of dynamically registered broadcasts.

Data Flow Analysis

`IntentFilter` is a parameter of broadcast registering methods, and `Intent` is a parameter of broadcast sending methods. Both `IntentFilter` and `Intent` are defined using action strings. `IntentFilter` can be initialized with an action string using `new IntentFilter(String action)` method, or it can be initialized first using `new IntentFilter()` method and later defined using the `addAction(String action)` method. We perform a backward data flow analysis on these methods using Soot so as to identify the required action strings. In total, we discover 130 unique broadcast action strings (238 instances) from system services and 207 unique action strings (424 instances) from system applications. After that, we determine whether the retrieved action strings are protected. By extracting protected broadcasts from our discovered broadcasts, we have 50 unprotected broadcasts in system services and 156 unprotected broadcasts in system applications, which can be abused by an attacker.

3.4 Attacking without Permissions

Our static analysis provides a list of unprotected APIs from system services, a list of unprotected components from system applications, and a list of unprotected dynamically registered broadcasts. We confirm the attacks by exploiting them with a third-party application without any permissions. In particular, we show that Java reflection attacks can be launched on APIs supported by system services and that intent-based attacks can be launched by exploiting the APIs supported by system applications and dynamically registered broadcasts. The attacks are performed semi-automatically: many codes used in the attacks are generated automatically, while parameters required for some attacks are identified manually. For instance, we automatically generate the codes, such as `sendBroadcast(new Intent("actionString"));`, where `actionString` is replaced by the real action strings discovered from our static analysis. Note that we aim not to provide an exhaustive list of all possible attacks but to show how easily serious attacks can be launched to Android smartphone systems without requesting for any permissions.

3.4.1 System Services

We identify two main types of possible attacks via the unprotected APIs supported by system services. An attacker may control various audio functions of mobile devices and steal users' information.

Audio Control

A system serivce, `IAudioService`, provides several unprotected APIs for controlling the audio systems. Without requesting for any permissions, an attacker may trigger call ringtones and alarms that users personally set for their phones. An attacker may produce other special sound effects, such as notification, click, and key-press sounds. To do so, an attacker first uses the `getRingtonePlayer()` API to obtain an `IRingtonePlayer` object, and then invokes its `play()` method. Moreover, an attacker may arbitrarily set the volumes of call ringtone, alarm, notification, music, system and voice call sounds using the `setStreamVolume()` API. An attacker abusing both unprotected APIs can be dangerous. For instance, an attacker may set the devices to their highest volumes and start playing ringtone or alarm sounds continuously. In such cases, even when devices are set to silent mode, they start ringing, which may disturb users in various social situations, such as in meetings. The only way for users to stop such attacks is to shut down their phones. In similar attacks, an attacker may confuse users by playing notification sounds without sending any notifications. Alternatively, an attacker may set the volume to 0 so that users become unaware of any incoming calls or alarms.

Leaked Information	Description	Exploited Method	Exploited Class
Device ID	Unique device ID, such as IMEI for GSM and the MEID or ESN for CDMA phones	getDeviceId()	IPhoneSubInfo
SIM Card State	Whether SIM card is ready, absent or requires PIN to unlock	getSimStateFor Subscriber()	iSub
Lock Setting	Whether user sets password or pattern lock	havePassword() and havePattern()	ILockSettings
Call state	Whether there is an incoming call, established telephony call, or established audio/video chat or VoIP call	getMode()	IAudioService
Ringtone mode	Whether ringer mode is silent and vibrate, silent and not vibrate or normal	getRingerMode Internal()	IAudioService
Input Device	External and internal input devices, such as joystick or keyboard type	getInputDevice()	IInputManager
Country	Current country of user	detectCountry()	ICountryDetector
Copied data	Copied data from clip board	addPrimaryClip ChangedListener()	IClipboard

Table 3.2: Information Leakage From System Services

Information Leakage

An attacker may obtain information about user's device ID, SIM card state, call state, ringer mode, input devices, country and copied data from clipboard. The unprotected APIs exploited for such attacks are shown in Table 3.2. Interestingly, we discover that device ID and SIM card state are accidentally made available via unprotected APIs, although they are supposed to be protected by `android.permission.READ_PHONE_STATE` permission. An attacker, who tracks such information continuously, can easily identify individual users and infer users' behaviours, which violates users' privacy.

3.4.2 System Applications

By exploiting unprotected APIs of system applications, an attacker may launch the following attacks: broadcast theft, malicious broadcast injection, activity hijacking, malicious activity launch, service hijacking, and malicious service launch. Broadcast theft, activity hijacking, and service hijacking attacks occur when an attacker intercepts intents by registering intent filters with unprotected action strings in its `AndroidManifest.xml` file. Malicious broadcast injection, malicious activity launch, and malicious service launch attacks occur when an attacker sends intents with `sendBroadcast(Intent)`, `startActivity(Intent)` and `startService(Intent)` methods. Such intents are initialized with unprotected action strings identified in the previous section. The intent theft attacks normally result in information leakage and component hijacking, while the other attacks result in unintended changes of Andriod system state.

Broadcast Theft

We discover that an attacker is able to obtain network, alarm, and account related information by intercepting unprotected broadcast intents. From the broadcast with action string `android.net.conn.CONNECTIVITY_CHANGE`, an attack may obtain network name, network state (e.g. connected, disconnected, connecting), network type (e.g. Wi-Fi or mobile LTE), and roaming status without requesting for `android.permission.ACCESS_NETWORK_STATE` permission. An attacker may receive `android.app.action.NEXT_ALARM_CLOCK_CHANGED` broadcast when a next alarm is set on the device. An attacker can also obtain `android.accounts.LOGIN_ACCOUNTS_CHANGED` broadcast when user account information (e.g., Gmail, Facebook, Skype account) is changed. Although some information leakage seems benign, it becomes serious when combined with other information. For instance, by constantly retrieving device ID and network name, an attacker may identify an individual user and determine the user's home and work locations.

Malicious Broadcast Injection

Various attacks can be launched by sending broadcasts with unprotected action strings. An interesting finding is that the action string `android.bluetooth.intent.DISCOVERABLE_TIMEOUT` is unprotected. By continuously sending broadcasts with this action string, an attacker can block other Bluetooth phones from discovering the exploited device. This attack disables the scan mode of the device's Bluetooth adapter and thus, makes its Bluetooth service unusable. Another finding is that by exploiting the action strings, `android.btopp.intent.action.OPEN_RECEIVED_FILES` and `android.intent.action.DOWNLOAD_NOTIFICATION_CLICKED`, an attacker may open the folders where the mobile user receives files from Bluetooth transfer, and where the downloaded files exist. Finally, an attacker may launch an input method chooser for different languages using action string `android.settings.SHOW_INPUT_METHOD_PICKER`.

Activity Hijacking

During activity hijacking, an attacker launches its own applications when intents with unprotected action strings are triggered. From our analysis, we discover that `android.intent.action.DIAL` and `android.media.action.STILL_IMAGE_CAMERA_SECURE` action strings are not protected by any permissions. Thus, an attacker may hijack phone and camera applications, when users launch them from their lock screens. Another finding is that an attacker may hijack Bluetooth, Wi-Fi, account (e.g. Gmail account), and Virtual Private Network (VPN) setting pages, when they are launched from the Setting application. The exploited action strings include `android.settings.BLUETOOTH_SETTINGS`, `android.settings.WIFI_SETTINGS`, `android.settings.ADD_`

ACCOUNT_SETTINGS, and android.net.vpn.SETTINGS. However, to trick users completely, the attacker have to implement full functionalities, which may requires permissions. Moreover, activity hijacking is hindered by the application chooser, which is launched, when there are conflicting applications handling the same intent. It is thus difficult for an attacker to launch these attacks without being noticed.

Malicious Activity Launch

Leaked Information	**Description**	**Exploited Broadcast Action String**
Network and Wi-Fi	NetworkInfo object - network name, network state (e.g. Connected, disconnected, connecting), network type (e.g. Wi-Fi or mobile LTE) WifiInfo object- SSID, BSSID, MAC address, link speed, frequency LinkProperties object - Interface name, link address, routes, DNS address, domains	android.net.conn.CONNECTIVITY_CHANGE android.net.wifi.STATE_CHANGE android.net.wifi.WIFI_STATE_CHANGED
Tether State	Which portable Wi-Fi hotspot is on and available	android.net.conn.TETHER_STATE_CHANGED
Airplane Mode	Whether airplane mode is on or off	android.intent.action.AIRPLANE_MODE
NFC State	Whether NFC is on or off	android.nfc.action.ADAPTER_STATE_CHANGED
SIM Card State	Whether SIM card state, such as ready or absent, changes	android.intent.action.SIM_STATE_CHANGED
Phone Service State	Whether phone is in service, out of service, emergency only or power off	android.intent.action.SERVICE_STATE

Subscription State	Whether data, SMS or voice subscription changes	android.intent.action.ACTION_DEFAULT_SUBSCRIPTION_CHANGED android.intent.action.ACTION_DEFAULT_DATA_SUBSCRIPTION_CHANGED android.intent.action.ACTION_DEFAULT_SMS_SUBSCRIPTION_CHANGED android.intent.action.ACTION_DEFAULT_VOICE_SUBSCRIPTION_CHANGED
GSM/CDMA Strength	Various measurements including LteRsrp, LteRssbr, LteCqi, CdmaDbm, CdmaEcio, GsmSignalStrength, EvdoDbm, EvdoSnr, EvdoEcio, GsmBitErrorRate	android.intent.action.SIG_STR
Location Mode	Whether location mode, such as high accuracy (use GPS, Wi-Fi, cellular network to determine location), battery saving (use Wi-Fi and cellular network to determine location) or device only (Use GPS to determine location), changes	android.location.MODE_CHANGED android.location.PROVIDERS_CHANGED
Volume	Volum value and whether phone is muted	android.media.VOLUME_CHANGED_ACTION android.media.RINGER_MODE_CHANGED android.media.STREAM_MUTE_CHANGED_ACTION
USB State	Whether USB is connected, in ADB mode or configured	android.hardware.usb.action.USB_STATE
Power State	Whether power is connected or disconnected	android.intent.action.ACTION_POWER_CONNECTED android.intent.action.ACTION_POWER_DISCONNECTED

Table 3.3: Information Leakage From Dynamically Registered Broadcasts

We discover that an attacker may launch several unprotected activities from system applications. Since some activities are entry points of system applications, this attack leads to the launching of the corresponding applications. An attacker may launch lock screen, emergency dialer, camera, mail, and music applications in such attacks. Some attacks, such as launching lock screen and emergency dialer, may confuse users, while other attacks, such as launching camera, may drain device batteries. In the following, we provide more details about such attacks for different system applications.

Warnings: An attacker may launch activities for the following warning messages: "Network monitoring: A third party is capable of monitoring your network activity, including emails, apps, and secure websites. A trusted credential installed on your device is making this possible.", "'Attention. You need to set a lock screen PIN or password before you can use credential storage," "Attention. Remove all contents? Cancel or Ok," "Oops! This device is already set up," and "To improve location accuracy and for other purposes, null wants to turn on network scanning, even when Wi-Fi is off. All this for all apps that want to scan? Deny or Allow." A severe consequence of these attacks is that user's selection from the warning messages takes real effect on the state of the phone.

Setting UIs: 67 activities of the system setting application are exposed to third-party applications in our findings. These activities include setting User Interfaces (UIs) for security (lock screen, encryption, credential storage and device administration), privacy (factory reset, restore and backup), developer options, Bluetooth, Near Field Communication (NFC) payment, Wi-Fi, location, sound, USB, and system notification. An attacker may launch these interfaces at any time without requesting for any permissions.

Others: Several hidden features can be launched using unprotected action strings. An example is the colour correction setting. When this activity is launched, the exploited interface states that this colour correction feature is experimental and may affect the performance of phones. Another attack is to launch the mobile emergency alert setting page, which lists the threats to life and property (e.g., robbery) around the area. Other unprotected activities includes the Wi-Fi network choosing interface, the brightness setting interface, the wallpaper setting interface, the live wallpaper choosing interface, and the downloaded file interface.

Service Hijacking

From Android 5.0 and above, only explicit intents with clearly stated package names can be used for binding services. Consequently, an attacker cannot launch any service hijacking attacks by simply declaring similar services with the same action strings as those of system applications' services.

Malicious Service Launch

There are two steps involved in launching the malicious service launch. An attacker first binds the services exposed from system applications and then invokes the methods inside. We discover that an attacker can successfully bind 15 services of system applications, including 14 services from Bluetooth system applications and one media service. An attacker may search for the class names of the exposed services in `AndroidManifest.xml` files of system applications, and use their class names for binding with explicit intents. After binding, however, an attacker cannot invoke any exposed methods from these services, because these methods are well-protected inside the source code of services. For instance, the methods from the Bluetooth system application are protected by the `android.permission.BLUETOOTH` and `android.permission.BLUETOOTH_ADMIN` permissions. Therefore, an attacker cannot launch any useful attacks by simply invoking these exposed methods.

3.4.3 Dynamically Registered Broadcasts

We show that several broadcast theft and malicious broadcast injection attacks can be launched by exploiting unprotected dynamically registered broadcasts.

Broadcast Theft

Similar to the broadcast theft attacks to system applications, an attacker may steal user information from unprotected dynamically registered broadcasts. We discover that an attacker is able to obtain network and Wi-Fi information, tether state, airplane mode, NFC state, SIM card state, phone service state, subscription state, GSM/CDMA strength, location mode, volume, USB state, and power state. A detailed description about the information leakage due to broadcast theft is given in Table 3.3. Although some of the leaked information seems benign, much useful information can be inferred from it. For example, location information can be inferred from Wi-Fi data and GSM/CDMA strength [64]. Users' payment and travel behaviours may be inferred from NFC state and airplane mode. Users' sleeping patterns can be inferred from USB state and power state [41]. We discover that some information is available to an attack application without any permissions, even though it is stated in Android API documentation that such information must be protected by permissions. For instance, according to Android API documentation, network and Wi-Fi information should be protected by permission `android.permission.ACCESS_NETWORK_STATE`; the SIM card state, phone state, and GSM/CDMS signal strength information should be protected by permission `android.permission.READ_PHONE_STATE`. This shows that dynamically registered broadcasts leak a lot of information to third-party applications, and platform providers should take additional steps to protect these broadcasts.

Malicious Broadcast Injection

An attacker may broadcast false information via malicious broadcast injections. We discover that intended receivers of these unprotected broadcasts are third-party applications or vendor-customized system applications. However, they are excluded from our study, as we focus only on Android framework as the attack target. Thus, although we have confirmed that an attacker can send these broadcasts, further analysis is required to study the impact of the attacks to third-party applications and vendor-customized system applications.

We discover that an attacker may send false commands for music applications, such as "next", "pause", "previous", and "toggle pause". The exploited action strings in this attack include `com.android.music.musicservicecommand.next`, `com.android.music.musicservicecommand.pause`, `com.android.music.musicservicecommand.previous`, and `com.android.music.musicservicecommand.togglepause`. Moreover, an attacker may send malicious information about the status of currently running music, such as its metadata, play state, and queue state. The exploited action strings include `com.android.music.metachanged`, `com.android.music.playstatechanged`, and `com.android.music.queuechanged` broadcasts. We also discover that an attacker may send broadcast `android.security.STORAGE_CHANGED`. This broadcast is triggered when (i) a new Certificate Authority (CA) is added, (ii) an existing CA is removed or disabled, (iii) a disabled CA is enabled, or (iv) the trusted storage is reset. This attack may cause serious problems to the receiving applications that act according to the received broadcasts. Moreover, an attacker may maliciously broadcast user log-in account, NFC state, data connection state, and emergency callback mode changes. The exploited action strings in these cases are `android.accounts.LOGIN_ACCOUNTS_CHANGED`, `android.nfc.action.ADAPTER_STATE_CHANGED`, `android.intent.action.PRECISE_DATA_CONNECTION_STATE_CHANGED` and `android.intent.action.EMERGENCY_CALLBACK_MODE_CHANGED`.

3.5 Attacking a Different Version

We apply our study to AOSP version 4.4.4_r1, and compare to what we have discovered on AOSP version 5.1.0_r1. We discover the differences in terms of unprotected APIs and viable attacks on these two versions. The attacks on version 4.4.4_r1 have been reported to the Google's security team, and most of them have been mitigated in version 5.1.0_r1. Even so, we still discover more unprotected APIs and new attacks in version 5.1.0_r1, which have also been reported to Google. This implies that the ad-hoc effort in mitigating the reported attacks is not sufficient, and systematic analysis would be helpful for platform developers to analyze unprotected APIs and improve the security of new AOSP versions.

3.5.1 Retrieving Unprotected APIs

We discover 79 system services and 69 system applications on AOSP version 4.4.4_r1. Compared to AOSP version 5.1.0_r1, we have 10 less system services, and the same number of system applications. Some system services, such as the fingerprint service and the web-view update service are not included in AOSP 4.4.4_r1. Our analysis reveals 557 unprotected APIs from AIDL interfaces of system services, which count for 34.77% of all 1,602 public methods on AOSP version 4.4.4_r1. There are 88 unprotected unique broadcast action strings (150 instances), 165 unprotected activity action strings (394 instances) and 18 unprotected service action strings (30 instances) from system applications in our results. It is also discovered that 47 out of total 114 dynamically registered broadcasts in the source code of system services are unprotected, and 124 out of 171 dynamically registered broadcasts in source code of system applications are unprotected. Compared to AOSP 5.1.0_r1, the number of unprotected APIs is smaller. This result is alarming, since it indicates that more unprotected APIs are introduced to the framework as new APIs are added in the later version.

3.5.2 Attacking without Permissions

Our attacks on AOSP previous version 4.4.4_r1 can be summarized as follows.

Denial-of-Service Attacks

By exploiting a single unprotected API, `Content.cancel Sync()`, an attacker may launch denial of service attacks on the synchronization of all content providers. This synchronization API is used for transferring data between an Android device and web servers. We discover that on Nexus 5, an attacker may block the synchronization of Gmail, Google Calendar, Google Drive, Google Note, Chrome and etc. By doing so, the attacker can prevent users from receiving new emails, even when users manually click on "refresh" in email apps. An attacker may also prevent synchronizing new calendar events with users' desktop computers, receiving newly shared google drive documents, and synchronizing Google notes, synchronizing Chrome's bookmarks, history, tabs, passwords, extensions and many browser-related information. Moreover, we discover that other popular applications, including Dropbox, Twitter, Facebook, Skype and Mozilla Firefox, also use the synchronization API `Content.cancel Sync()`. For instance, Skype uses the API for synchronizing contact information, while Firefox uses it for synchronizing bookmarks, history, tab and password information. Thus, their synchronization functions can be deferred by an attacker.

Other Attacks

An attacker may send notifications to users, set car mode (which is used to open speaker directly from calls), set night mode (which allows the OS to intelligently change the color theme depending on the time of day), wake up the device at certain time (without the wake-lock permission), and set the screen-off time. Moreover, an attacker may obtain a variety of valuable information from users' devices, including what password salts are used, whether users set security for lock screen, whether users use passwords, pins or pattern locks for log-in, whether the lock screen is on or off, and whether the screen is turned on. Moreover, a false system notification can be sent to show that devices have entered into the emergency callback mode.

3.6 Discussions

Our research reveals many attacks that can be launched by applications with no permissions. This discovery is important, because a significant portion of Android security research focuses on applications that have permissions, and no one has looked into the unprotected resources, which are easily accessible without permissions. Many of our attacks, after being reported twice for two versions, have been acknowledged and fixed by the platform provider. This shows that our analysis on unprotected APIs is necessary in improving the security of Android framework. Note that we do not suggest to reclassify and protect all the corresponding resources that are attacked in this work. The reason is that protecting all resources may degrade the usability of the framework. For example, usability researchers state that too many permission requests may cause users to grant permissions without careful considerations [28]. Therefore, while we highlight the security flaws of unprotected APIs in this work, we believe that an optimal defense mechanism should consider not only the security and privacy aspects but also the flexibility and usability aspects of the framework. Coming up with an optimal solution for this problem is not trivial and requires involvement from both research and industry communities. Thus, we leave it as future work to find various ways of protecting the currently unprotected resources without degrading other aspects of the framework. In the meantime, we suggest platform providers to systematically analyze unprotected APIs before releasing new versions, so that similar attacks are prevented in the future.

We identify three ways in which our analysis of Android APIs can be improved and used as a commercial vulnerability analysis tool. First, our work focuses only on detecting unprotected APIs and exploiting them for attacks. Thus, a natural step forward is to determine whether an unprotected API is indeed vulnerable by analyzing the nature of the API source code. Second, platform providers may consider analyzing other types of unprotected APIs, such as callback methods, listeners and class fields. Callback methods and listeners provide alternative ways of inter-process communication, and they may expose some vulnerabilities from Android APIs. Third, platform providers may consider a more advanced adversary, which possesses certain privileges or permissions. Such adversary may be categorized according to its permission level, such as `normal`, `dangerous`, `signature`, and `signatureOrSystem`, and/or according to the types of Linux users associated to privileges, such as `root`, `system`, `keystore`, `media`, `nobody`, `wifi`, and `u0_a86`. A more advanced adversary would lead to more serious attacks.

3.7 Related Work

The mapping between API calls and permission checks on Android has been investigated in prior research, including Stowaway by Felt et al. [27], COPES by Bartel et al. [9] and PScout by Au et al. [6]. Our work is different from these works in that they focus on permission usage, while our work focuses on unprotected APIs and potential exploits. Besides permission checking, we also consider Linux ID checking in our analysis.

The topic of system-level vulnerabilities in Android framework has been studied before. DexDiff by Mitchell et al. [61] investigates the vulnerabilities in vendor-customized frameworks by comparing them with the official Android systems in binary analysis. ADDICTED by Zhou et al. [102] performs the analysis of vendor-customized components. In particular, it identifies critical Linux files and compares their protection levels in terms of Linux file permissions between customized framework and AOSP. If a file is less protected on customized framework, then it is more likely to be attacked. This line of works focuses on the vendor-modified components of Android frameworks, while we focus on unprotected APIs and their exploits in

Android frameworks. Similarly, Wu et al. [87] study vendor customizations of system applications. They discover that the vendor-customized applications are vulnerable to permission re-delegation attacks, confused deputy attacks, passive content leak attacks, and content pollution attacks. Another way of Android vulnerability analysis is performed by Yang et al. in IntentFuzzer [90] and Ye et al. in DroidFuzzer [93]. They automatically construct intents and use brute force to discover vulnerabilities. However, as stated in IntentFuzzer, their research does not penetrate deep into application logic nor uncover interesting bugs for launching serious attacks. Kratos [78] also uses call graph analysis to find vulnerabilities in Android framework. However, it focuses on the inconsistencies of security checking, while our work focuses on APIs without any security checking.

Vulnerable components of third-party applications have been investigated before. ComDroid by Chin et. al. [15] analyze the inter-application communications among third-party applications so as to identify vulnerable components of third-party applications. We use a similar threat model in our analysis and apply it on system applications. Wu et al. [86] use the reachability analysis to identify and categorize such vulnerabilities. Another work, EPICC, by Octeau et al. [68] show that over 93% of third-party applications contain vulnerable components. To exploit the vulnerable components, Li et al. [51] propose an approach which can automatically generates an attack application. In comparison to these works, part of analysis in our research focuses on intent-based vulnerabilities of system services and system applications, rather than third party applications. On the other hand, a line of work focuses on how to prevent attacks from exploiting vulnerable components of third-party applications. For example, CHEX by Lu et al. [55] mitigates such attacks by statically vetting third-party applications. AppSealer by Zhang and Yin [94] focuses on how to generate security patches automatically so as to prevent the intent-based attacks to vulnerable components of third-party applications. Oh et al. [69] propose a solution to address the denial of service attacks on ordered broadcast intents.

Another line of work focuses on how to better manage the security of Android frameworks. Heuse et. al [42] survey different research solutions related to Android access control mechanisms and propose a new general programmable interface, called Android Security Module (ASM), which can be used for managing access control policies. Likewise, Backes et al. [7] propose an Android Security Framework (ASF), in which the access control policies for Android APIs can be modified or extended easily by enterprises or security experts without modifying the framework. Complementary to these works, our work focuses on unprotected APIs and potential attacks, which could be mitigated by placing appropriate access control mechanisms.

3.8 Conclusions

In this chapter, we show how an attacker, which is a third-party application without any permissions, can attack Android smartphone systems by exploiting various unprotected Android APIs, including unprotected AIDL interfaces of system services, unprotected components of system applications, and unprotected dynamically registered broadcasts. The attacks we discover include blocking Bluetooth and email services, controlling audio functions, stealing valuable device information, and hijacking system activities and broadcasts. The result of this work suggests that Android platform providers should carefully analyze the exposed APIs, and mitigate any identified attacks. We envision that with more features added to Android devices, larger source code sizes of Android frameworks, and faster paced releases of Android versions, such analysis and mitigations are much needed to achieve better security in Android system development.

Chapter 4

Dissecting Policy-Violating applications: Characterization and Detection

4.1 Introduction

Google Play store is infected with various types of bad applications, including malware applications, privacy-violating applications, and repackaged applications. Google has been trying to take down these applications everyday with the help of anti-malware technologies, such as Google Bouncer, and inputs from researchers and users. Previously, researchers have tried to understand malware applications' behaviors by painstakingly collecting real-life malware application samples [103]. They have also proposed various prevention or detection techniques for malware applications [36] [1] [72], privacy-preserving applications [92] [106] [91] [88] [58] [82], and repackaged applications [96] [101] [79] [14] [33] [98]. However, many applications behave in a manner that is undesirable, and yet less serious than these applications. For instance, some applications redirect users to share about the applications on Facebook page. Some are spams. Some applications simply lack proper functionalists and qualities. These applications violate Google Play developer policies but little has been discovered about the overall picture of these applications. Therefore, we attempt to answer the following questions in our paper: What categories of bad applications are the most common ones on Google Play store? What characteristics and behaviors make them reported by users and removed from the market? Can the existing anti-virus solutions be used in detecting them?

Our paper makes the following contributions: (1) We collect a set of real-life applications that are reported by users, and later taken down by Google Play store. The lack of sample set has been deterring researchers from creating solutions and evaluating their effectiveness. Our collection of real-life bad applications can serve as a baseline for various future analysis and research (2) We perform extensive empirical analysis on the application samples we collected. Comprehending and characterizing these applications is the first step towards designing defense mechanisms against them. Our empirical analysis provides answers to various questions left unanswered by previous research (3) We use machine learning approach to detect the collected application samples. Although machine learning algorithms have been commonly applied for malware applications detection, different feature sets are required for detecting our application samples due to their unique characteristics and behaviors. In our paper, we also figure out whether the existing anti-virus solutions can effectively detect these policy-violating applications.

First, we build an automated crawler, which collects real-life applications that violate Google Play developer policies. Google enforces these policies to maintain quality and health of mobile ecosystem as well as to provide great experience for mobile users. Our crawler crawls the posts from Reddit forum under Bad application category every 5 minutes. It obtains the links of reported applications from the posts and immediately downloads them from the Google Play store. After 3 months of automated crawling, our crawler follows the same links again and checks if the bad applications are indeed removed from the official Google Play store. In this way, we are able to collect 302 bad applications that are reported and removed from the Google Play store.

Second, we perform extensive empirical analysis on the application samples we collected. Out of 302 crawled applications, we discover that 161 applications violate intellectual property rights of other applications or brands. The most copied applications include Bejeweled Blitz, Candy Crush, Mine Craft, Angry Bird Rio, Flappy Bird, Hay Day, Fruit Ninja, Subway Surf, Construction City, Sonic Dash, Gangster Vegas, and Grand Theft Auto. The most copied trademarks or brands include Pikachu, Adobe, Pou, Mario, Disney, Mickey, Minion, Counter Strike, and Despicable Me. The violation normally takes place in applications' titles, descriptions, icons or screen-shots. We discover that 79 of them use similar titles, 76 use screen-shots that are different from in-app screen-shots, and 73 use mis-leading descriptions that are different from applications' actual functions. Many reported bad applications use misleading keywords, such as 2, II, Demo, Free, Pro, 3D, and HD, claiming that they are an enhanced version of original applications. Moreover, we discover 67 applications that claim to contain certain functions but actually do nothing. They include fingerprint scanner applications, flash light applications, font applications, wallpaper applications, bluetooth applications, volume boosters, wifi boosters, and mp3 downloaders.

Among the crawled applications, 147 applications violate ad policies. We discover 70 applications with ads that simulate the user interfaces of the applications, 54 applications that modify browser settings or add homescreen shortcuts on the users' device as a service to third parties or for advertising purpose, 34 applications that show ads outside the applications, 17 applications from which users cannot dismiss their ads without penalties or inadvertent click-throughs, and 10 applications that display ads through system level notifications. Our analysis shows that the most common violating ad libraries include startapp, inmobi, umeng, ironsource and actionbarsherlock. Moreover, we discover that 30 applications redirect users to install other applications from Google Play store or third-party markets, 16 applications violate Youtube policies by downloading videos from Youtube, and 9 applications download unwanted mp3 files. We also discover 2 applications which are automatically created by wizard services, and another 2 applications which offer incentives to rate the applications.

Third, we apply machine learning algorithms for detecting such misbehaving applications. To test the effectiveness of our detection, we apply it on 302 real-life policy-violating applications that we collected and 326 benign applications from Google Play store. We extract 175 features from the applications' use of brand names and keywords, third-party libraries, network activities, meta data, permissions, and misbehaving API calls that are originated from third-party libraries. We input the extracted features into 10 machine learning classifiers for differentiating policy-violating applications from benign applications. Three-fold cross validation is performed, where two-third of our data set is used for training and one-third of our data set is used for testing. The experimental results show that our detection algorithm can effectively detect bad applications with 86.80% true positive rate and 13.6% false positive rate. We also test our samples with VirusTotal [84], which scans the submitted applications with existing 57 anti-virus solutions. We assume that VirusTotal can detect a policy-violating application as long as one of its anti-virus software reports it as bad. In this way, we discover that the true positive rate of VirusTotal is 55.63% and its false positive rate is 17.48%. In terms of individual performance, the best anti-virus solution of VirusTotal can detect the policy-violating applications with the true positive rate of 36% only. Our research shows that despite the efforts of industry and research communities in application market regulation, the problem of policy-violating applications is still prevalent and requires attention from both industry and research communities.

The rest of the paper is organized as follows. Section 4.2 clarifies our data collection process. Section 4.3 provides our empirical analysis and findings. Section 4.4 describes the details of our detection mechanism, and Section 4.5 discusses the experiment results. Section 4.6 summarizes the related work and Section 4.7 concludes the paper.

4.2 Data Collection

In this section, we describe how we collect policy-violating application samples. These samples are very useful for analyzing the applications' behaviors and consequently for designing defense mechanisms and evaluating them. However, obtaining a set of policy-violating applications is not trivial. There are two challenges to it. The first challenge is noticing what applications are violating Google play policies, because users' reports to Google Play store are not available to the public. To solve this, we seek to a public forum, Reddit, for such application reports by users. The second challenge is that there is a small time frame to crawl or download policy-violating applications from Google Play, once the report has been made. To solve this, we develop an automated crawler for downloading bad applications and creating a database of bad applications. We plan to make the dataset available to the public, after this paper is published.

To find users' reports, we first crawl posts from Reddit forum under the `https://www.reddit.com /r/Badapplications` URL. This URL is a subReddit (i.e. sub-forum) used to report inappropriate applications to Google. The description of the subReddit is as follows. "A subreddit to discuss and coordinate reporting bad applications to the Google Play Store. Note: A bad application refers to applications that are fake, pretending to be from different developer, harmful, are there just to serve annoying ads to you, or steal your info. An application that is just poorly made should not be posted here, they are fine." Reddit provides Application Programming Interfaces (APIs) for various functions, including messaging, editing posts and reading posts. The returned objects are in JavaScript Object Notation (JSON) format. We develop a Python crawler, which checks the BadApp subReddit forum every 5 minutes for latest users' posts and comments. Once we find a new link of reported app, we crawl the applications' metadata, as well as Android Application Package (APK) files from the Google Play store. The crawling continues for over 3 month period. After that, we check the same links of the applications again, and see if they are indeed removed from the Google Play store. In this way, we crawl about 302 policy-violating applications.

The second set of our data is benign applications from Google Play store. We randomly download 326 benign applications, which exist for at least 3 months from the Google Play store with an unofficial Python API.

4.3 Empirical Analysis

In this section, we perform empirical analysis on our bad application samples and explain our findings on their characteristics and behaviors. To understand their behaviors, we first put the Google Play's developer policies into four main categories: (1) intellectual property and deception, (2) monetization and ads, (3) spam, store listing and promotion, and (4) security and privacy. Although restricted content, such as sexually explicit content and violence, are parts of Google Play policies, we do not find any applications violating these policies. Table 4.1 shows the results of our empirical analysis. We discover that more than half of bad applications are violating copy-rights or trademarks and about half of bad applications are violating ad policies. Many applications show several other bad behaviors, such as spamming, violating Youtube policies and downloading external files. The most common types of policy violations are (i) applications that use similar title to branded applications, (ii) applications that use photos or screenshots from other brands, (iii) applications that provide misleading description, (iv) applications that have little or no functions, and (v) applications that hold ads simulating the user interfaces.

Category	**Policy Violating Behaviors**	**No of apps**
Icon (Copy-right)	Same as the icon of original app	4
	Similar to the icon of original app	15
	Violates copyright or trademark of original app (e.g. copying title, screenshots, in-app real screenshots of original app)	52
	Violates copyright or trademark of other brands or websites	32

Title (Copy-right)	Same as the original app	3
	Similar to the original app	79
	Violates copyright or trademark of other brands or websites	35
Screenshot (Copy-right)	Same as the original app	35
	Similar to the original app	4
	Violates copyright or trademark of original app (e.g. copying title, in-app real screenshots)	39
	Violates copyright or trademark of other brands or websites	38
	Different from the real in-app screen	76
Description (Copy-right)	Same as the original app	5
	Similar to the original app	36
	Violates copyright or trademark of original app (e.g. including other apps brand names)	41
	Violates copyright or trademark of other brands or websites	38
	Different from actual function (Misleading descriptions)	73
	Irrelevant and excessive keywords in apps descriptions	22
Function (Copy-right)	Very little or no function (e.g. blank page or a video keeps playing)	67
	Same as the original app	6
	Similar to the original app	5
	Violates copyright or trademark of original app (e.g. including other apps resources)	7
	Violates copyright or trademark of other brands or websites	19
	The apps primary function is to reproduce or frame someone elses website (i.e. web-view of anther website)	19
Ads	Ads simulate or impersonate the user interface of any app	70
	Ads are displayed outside the app (Ads simulate or impersonate UI notification and warning elements of the operation system)	34
	Displays advertisements through system level notifications (Push notifications)	10
	Users cannot dismiss the ads without penalty or inadvertent click-through (e.g. exit ads)	17
	Modifies or adds browser settings or bookmarks, adds homescreen shortcuts, or icons on the users device as a service to third parties or for advertising purpose	54
Spams	Created by an automated tool or wizard service and submitted to Google Play by the operator of that service on behalf of other persons	2
	Offer incentives for rating	2

Others	Violates Youtube policies	16
	Automatically redirects users to install other apps (including other third-party market apps) from Google Play	30
	Includes buttons in apps to download other apps from Google Play	49
	Forces users to download files or install apps from sources outside of Google Play	25
	Downloads unwanted mp3 files	9

Table 4.1: Empirical Analysis on Policy Violations

Intellectual Property Right Violations

The first five categories of policy-violating behaviors in Table 4.1 are related to copy-right or trademark violations. Copying may occurs in five places of bad applications: icons, titles, screenshots, descriptions or functions. An interesting finding is that more copying applications make their icon, title and descriptions similar to those of original applications, instead of copying the exact features. Further analysis shows that reported applications tend to use the keywords, such as 2, II, Demo, Free, Pro, 3D, and HD, in their titles claiming that they are an enhanced version of original applications. Moreover, we discover that more bad applications make their icons from the screenshots included in the metadata of original applications or in-app real screenshots of original applications. However, many of them use the same screenshots that as the original applications. These findings have several implications for detecting applications that violate intellectual property rights. For instance, similarity scores between icons of copying applications, and screenshots of copied applications should play an important role in detecting such applications. We also discover that many of them uses screen-shots that are different from in-app screen-shots and mis-leading descriptions that are different from applications' actual functions. This suggests that if we can find discrepancies in these bad applications, we will be able to detect them well.

We discover that there exist not only applications which copy other original applications but also applications which violate the copy-rights or trademarks of other brand or websites. For instance, some applications include intellectual properties of Pokemon or Play Station although these original companies do not have any related applications in the Google Play store. We further analyze applications and brands that these bad applications normally copy. They include Bejeweled Blitz, Candy Crush, Mine Craft, Angry Bird Rio, Flappy Bird, Hay Day, Fruit Ninja, Subway Surf, Construction City, Sonic Dash, Gangster Vegas, and Grand Theft Auto. The brands or trademarks that are violated include Pikachu, Adobe, Pou, Mario, Disney, Mickey, Minion, Counter Strike, and Despicable Me.

Interestingly, we only find 11 repackaged applications whose functions are exactly the same or similar to original applications. This shows that although bad applications are one of the main distribution channels of malware, they are only a small portion of entire policy-violating applications. applications that claim to contain some functions but actually doing nothing include fingerprint scanner, flash light applications, font applications, wallpaper applications, bluetooth applications, volume boosters, wifi boosters, MP3 downloaders, and other music downloaders. The primary functions of some bad applications are to redirect users to other websites. We also find 22 applications which use irrelevant and excessive keywords in application descriptions. For example, some font applications use "Samsung Galaxy" keyword extensively in their descriptions to direct mobile users to their applications during the search. Such applications mostly reduce the quality of the application market.

Ad Policy Violations

The past studies on ad libraries have been focusing on privacy issues, such as phone ID and location collection by ad libraries. However, our empirical analysis results show a different set of policy-violating ad libraries. Our data set includes 70 applications with ads which simulate or impersonate user interfaces of the applications. We also discover 54 ad-policy violating applications that modify browser bookmarks or add homescreen shortcuts on users' mobile phones. Moreover, we find 34 applications, which display ads outside the applications using warning elements of Android. We also discover 17 applications where users cannot dismiss ads without penalty or inadvertent click-through and 10 applications where ads are sent via system notifications.

The most common policy-violating ad libraries are Start App, Inmobi, Umeng, and IronSource. Start App library shows interstitial ads, splash ads, exit ads, native ads and reward ads. Inmobi ad library includes interstitial ads and native ads. Umeng ad library sends ads via system notification bar, downloads and requests installation of new applications, and send information to a remote location about currently running applications, installed applications, and device information such as International Mobile Station Equipment Identity (IMEI), kernel version, phone manufacturer, phone model details, location (such as GPS coordinates, cell tower location), and network operator information. Ironsource library displays native ads and video ads.

In addition to ad libraries, we discover other common types of libraries among bad applications, including (i) `com.unips`, which provide live wallpaper adware, (ii) `com.monotype`, which claims to provide free font, (iii) `com.rahul`, which provides Youtube downloader, (iv) `com.unity3d` and `org.andengine`, which are game developing libraries, and (v) `io.card` credit card scanning library under the URL https://github.com/card-io/card.io-Android-SDK.

Spams

Some applications violate policies by applying automated application creation tools, such as `https://www.applicationsgeyser.com/`, or offering incentives for rating the applications. We find only 2 applications in each category. Despite this small number, we notice during crawling that some Reddit-forum users report developers who are spamming the market by creating hundreds of similar applications. Studying these spammers requires different kind of analysis on developer accounts in addition to applications themselves. Thus, we leave them as future work.

Others

We discover other applications that misbehaves, but Google's developer policies do not cover. They include applications that violate Youtube policies, redirect users to other applications, and force users to download unwanted files. We discover that 25 applications force users to download `mobogenie.apk` file, which enables control from a remote computer. Mobogenie may also be used to download applications, images, videos or music to mobile phones, manage SD cards, create backups on computer, and edit phone contacts. We find 30 applications, which redirect users to `spearmintbrowser.com`, which claims to provide AdBlock and build-in Flash support features. We also discover 10 applications, which download APK file named as flash player, 80 bad applications, which redirect users to download other applications from Google play and other third-party markets, and 12 applications, which connect to and grab data from Youtube. Many applications also access mobile users' accounts by obtaining authentication tokens: 118 applications access to Google account, 8 to PayPal account and 24 to Twitter account. Moreover, we find out that 29 bad applications attempt to share posts on users' Facebook account using URLs, such as https://m.facebook.com/dialog/feed?app_id={0}&link={1}&picture={2}&name={3}&description={4}&redirect_uri={5}.

4.4 Detection

Our detection includes two steps: (1) extracting typical features from both bad applications and benign applications and (2) applying selected machine learning algorithms to detect bad applications. The detection can be performed by either security researchers or Google Play Store managers for vetting submitted applications before they are officially released.

4.4.1 Feature Extraction

Classes	Methods
`android.app.NotificationManager`	`notify()`
`android.app.AlertDialog.Builder`	`show()`
`android.widget.Toast`	`show()`
`android.provider.Browser`	`saveBookmark()`
`android.provider.Browser`	`sendString()`
`android.content.Context`	`startActivity()`
`android.net.Uri`	`parse()`
`java.lang.ClassLoader`	`loadClass()`
`java.lang.Class`	`forName()`
`java.lang.Class`	`getDeclaredMethod()`
`java.lang.Class`	`getMethod()`
`java.lang.reflect.Method`	`invoke()`
`dalvik.system.PathClassLoader`	`init()`
`dalvik.system.DexClassLoader`	`init()`
`dalvik.system.DexFile`	`loadDex()`

Table 4.2: APIs Used as Features in our Detection

We extract six groups of features from mobile applications, including the use of brand names, third-party libraries, network activities, meta data, permissions, and API calls. The features can be grouped into two. The first category includes features derived from our empirical findings, such as popular brands or application names, network activities and third-party libraries. The second category is based on behavior-based features, such as permission and API-based features. Our feature extraction is implemented in Python, and detection algorithms are run in Java. In particular, Androguard library [21] is used to reverse engineer the application codes, and extract the information about third-party libraries, network activities, permissions and API calls. To obtain third-party libraries, we first use the `dx.get_tainted_packages().get_packages()` method of Androguard to obtain package lists from applications and then, we exclude the package names of the applications. To analyze the network activities, we first obtain the String values of APK files via the `d.get_strings()` method, and then, filter out the values starting with "`http://`" or "`https://`". For searching the API calls from third-party libraries, we apply `dx.tainted_packages.search_methods()` method, and determine whether they are originated from application source codes or third-party libraries' source codes.

Use of Brand Names and Other Keywords

We blacklist a list of popular brands and application names that we have obtained from our empirical analysis. Each brand name represents a feature for our detection. We also include other keywords, such as 2, II, Demo, Free, Pro, 3D, and HD as features. Overall, we use the following 56 words as features in our detection: 'flash', 'light', 'bejeweled', 'blitz', 'wAsk', 'racing', 'live', 'wallpaper', 'construction', 'city', 'studio', 'candy', 'bluetooth', 'free', 'game', 'sniper', 'crime', 'craft', 'mine', 'pikachu', 'font', 'farm', 'app', 'video', 'download', 'tube', 'pou', 'gangster', 'bird', 'flappy', 'subway', 'surf', 'dash', 'grand', 'theft', 'sonic', 'rio', 'ninja', 'demo', 'pro', '3D', '2', 'II', 'hay', 'day', 'flv', 'adobe', 'install', 'despicable', 'font', 'galaxy', 'monotype', 'volume', 'boost', 'mp3', and 'music'.

Third-Party Libraries

We blacklist the following 16 third-party ad libraries that are shown to include aggressive ad behaviors: ’startapp’, ’inmobi’, ’umeng’, ’ironsource’, ’actionbarsherlock’, ’millennial-media’, ’adsdk’, ’revmob’, ’chartboost’, ’fmod’, ’furry’, ’mobclix’, ’appflood’, ’tapjoy’, ’jirbo’, and ’squareup’. The presence of each library is regarded as one feature in our detection.

Network Activities

We maintain the following 20 blacklisted servers, and determine whether applications connect to them in their APK files: ’admob’, ’gstatic’, ’startappexchange’, ’ad-market’, ’search-results’, ’inmobi’, ’umeng’, ’googleapis’, ’akamaihd’, ’applicationsdt’, ’spearmint-browser’, ’mobilecore’, ’avazutracking’, ’cloudfront’, ’youtube’, ’rightyoo’, ’iron’, ’scm-pacdn’, ’airpush’, and ’ytimg’.

Meta Data

From the metadata of an app, we extract the number of downloads, the app’s APK file size, the number of ratings, the average star rating, the number of users rating one star, the number of users rating two star, the number of users rating three star, the number of users rating four star and the number of users rating five star. Thus, the meta data contribute 9 features for our detection.

Permissions

We use the the following 20 permissions as features in our detection. The first six permissions are derived from our empirical analysis of bad applications, which shows that many reported applications ask for credentials, contact list and hardware control, such as camera, audio or video. The next six permissions, such as `INSTALL_SHORTCUT` and `WRITE_HISTORY_BOOKMARKS`, are relevant to adware behaviors. The rest of the permissions, such as `BILLING` and `SEND_SMS`, are relevant to malware behaviors.

- android.permission.USE_CREDENTIALS
- android.permission.READ_CONTACTS

- android.permission.RECORD_AUDIO
- android.permission.CAMERA
- android.permission.CAPTURE_VIDEO_OUTPUT
- android.permission.CAPTURE_SECURE_VIDEO_OUTPUT
- android.permission.ACCESS_FINE_LOCATION
- com.android.launcher.permission.INSTALL_SHORTCUT
- android.launcher.permission.INSTALL_SHORTCUT
- com.android.browser.permission.READ_HISTORY_BOOKMARKS
- com.android.browser.permission.WRITE_HISTORY_BOOKMARKS
- android.permission.WRITE_SETTINGS
- android.permission.INTERNAL_SYSTEM_WINDOW
- android.permission.BILLING
- android.permission.SEND_SMS
- android.permission.CALL_PHONE
- android.permission.PROCESS_OUTGOING_CALLS
- android.permission.INSTALL_PACKAGES
- android.permission.RECEIVE_BOOT_COMPLETED
- android.permission.WRITE_EXTERNAL_STORAGE

API Calls

We identify 15 API calls that are required to complete specific behaviors of bad applications. Table 4.2 shows the list of API calls used in our detection. The first six API calls are required for adware behaviors, such as sending ads as notifications and changing browser settings. The rest of the APIs are used for Java reflection and dynamic code loading, since they are normally used by malicious applications to avoid being detected in static analysis. We extract 3 features from each API calls: presence of identified API calls, number of calls and whether the class files making the API calls are obfuscated. In total, we extract 45 features from API calls.

Others

There are other 4 types of features that we use for our detection. The first feature is whether the applications import cryptographic package `javax.crypto` because many malicious applications are known to encrypt and decrypt their codes. Another feature is whether the application files overwrite the `onBackPressed()` method, since this API is called by applications with exit ads. We also determine whether the APK codes include suspicious strings, such as `com.android.launcher. action.INSTALL_SHORTCUT`, since homescreen shortcuts may be added via intents with the above action strings. The final feature is whether applications include any string literals ending with ".apk", because many bad applications force users to download and install external APK files. Similar to API calls, we derive 3 types of information for each feature: presence of identified API calls, number of calls and whether the class files making the API calls are obfuscated

Algorithm	**True Positive**	**False Positive**	**Precision**	**Recall**	**F-Measure**
Naive Bayes	0.780	0.216	0.785	0.780	0.780
Logistic regression	0.774	0.228	0.774	0.774	0.774
SMO	0.868	0.136	0.871	0.868	0.867
Lazy-Ibk	0.828	0.177	0.833	0.828	0.827
Random Committee	0.855	0.148	0.857	0.855	0.855
Decision Table	0.750	0.256	0.754	0.750	0.748
PART	0.812	0.190	0.812	0.812	0.812
J48	0.815	0.188	0.817	0.815	0.815
LMT	0.854	0.151	0.857	0.854	0.853
Random Forest Tree	0.852	0.152	0.855	0.852	0.851

Table 4.3: Evaluation of Our Detection Algorithm

4.4.2 Detection

We apply 10 commonly used machine learning classifiers using Weka library [38]. The classifiers include Naive Bayes, Logistic, Sequential Minimal Optimization (SMO), Lazy-Ibk, Random Committee, Decision Table, Decision Part, J48, Logistic Model Tree (LMT), and Random Forest Tree. Naive Bayes is a family of simple probabilistic classifiers based on the Bayes' theorem. Logistic classifier applies the regression model, SMO applies Support Vector Machines (SVM), and Lazy - Ibk classifier applies K-nearest neighbours algorithm.

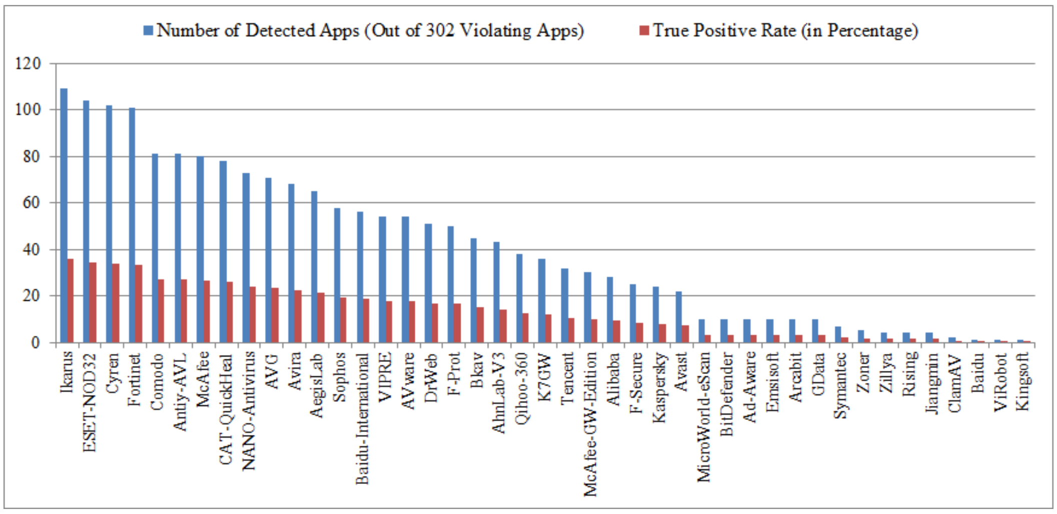


Figure 4.1: Detection Result by Anti-Virus Software from VirusTotal

Random Committee classifier uses a group of base classifiers, and the result is the average of the predictions generated by the individual base classifiers. Decision Table classifier uses a simple decision table. J48 Tree classifier, LMT classifier and Random Forest Tree classifier are algorithms based on decision trees.

4.5 Evaluation

In our evaluation, we apply the 10 classifiers to 628 applications used for our evaluation, including 302 reported bad applications and 326 benign applications. The 3-fold cross-validation is used for reducing over-fitting in our evaluation, where two-third of our data set is used for training and one-third is used for testing. Table 4.3 shows the weighted average of true positives, false positives, precision, recall and f-measures of our detection algorithms. True positive rates indicate the percentage of real malicious bad applications among the reported bad applications. False positive rates indicate the percentage of bad applications that are reported but are not really malicious. Precision is the ratio of true positives to true positives plus false positives. Recall is the ratio of true positives to true positives plus false negatives. F-measure combines precision and recall, and can be used as an overall measure for evaluation. For true positive rate, precision, recall and f-measure, the higher the scores are, the better the algorithm performs. The reverse is true for false positive rate.

In terms of f-measure, SMO classifier performs the best, followed by Random Committee and LMT classifiers. SMO classifier uses SVM, which is known to be the best classifier in many general cases. At the same time, Random Committee and LMT classifiers are good at dealing with binary and multi-class target variables, numeric and nominal attributes as well as missing values. Thus, they seem to be well-suited for our data set. The least effective classifiers are Naive Bayes, Logistic regression, and Decision Table classifiers. This is mainly due to the fact that Naive Bayes and Logistic regression classifiers perform the best for categorical dependent variables, while our dependent variables include non-categorical features, such as star ratings and numbers of downloads. Moreover, simple decision table classifier may not capture the complex rules of our feature set. We expect our results to be improved by focusing on the individual types of policy-violating applications. By doing so, we can select the features based on the behaviors specific to the types of applications. For instance, we can apply text and image similarity features for detecting intellectual property right violating applications. We leave this as future work, since the purpose of this paper is to characterize and detect all policy-violating applications.

We also compare our result with VirusTotal, which scans the uploaded applications with 57 anti-virus software. We submitted our 302 policy-violating applications to VirusTotal an retrieved the scanned report. Overall scan report shows that VirusTotal can detect only 168 of the submitted applications with its 57 anti-virus software. The remaining 134 applications are never alerted by any of the anti-virus software. At the same time, VirusTotal falsely reports 57 benign applications as policy-violating applications. Thus, we can say that the true positive rate of VirusTotal is 55.63% and the false positive rate of VirusTotal is 17.48%. The numbers of reported policy-violating applications (i.e. true positives) by individual anti-virus solutions are shown in Figure 4.1.

4.6 Related Work

Although there are limited studies on intellectual property right violating applications, there are many studies on the repackaged applications. Repackaged applications are the clones created from the reverse-engineered codes of original applications. Balanza et al.[8] analyze a repackaged malware, called DroidDreamLight, and Jung et al. [47] launch repackaging attack on bank applications. Chen et al. [13], and Gibler et al. [34] study the impact of repackaged applications and find out that 14% of original developers' revenues and 10% of user are redirected to the attacker. Potharaju et al. [74] use permission information and estimate that 29.4% of applications are likely to be plagiarized.

Since repackaged applications contain similar source codes as original applications, their detection mechanisms focus on the source code similarities. DroidMOSS [101] and Juxtapp [40] [52] apply fuzzy hashing on program instruction sequence and derive the similarity score by calculating the edit distance between two generated fingerprints. Crussell et al. [**?**] propose DNADroid, which uses Program Dependence Graph(PDG) to determine code similarity. AnDarwin [17] applies Locality-Sensitive Hashing (LSH), Lin et al. [54] use thread-grained system call sequences and Zhou et al. [100] propose linearithmic search algorithm in a metric space to detect repackaged applications. Deckard [46] uses a tree-based detection algorithm for detection. Huang et al. [43] propose an evaluation framework for detection algorithms of repackaged applications by measuring their resilience to obfuscation methods. Different from other approaches, Zhou et al. [99] propose to use software watermarking to prevent repackaging. Since these methods only focuses on code similarity, they cannot detect applications, which copy the external features of original applications and not their source codes.

Similar to our paper, several previous work highlights various issues of ad libraries. However, they focus more on privacy, security and usability issues, and not on their aggressiveness for showing ads or obtaining clicks from users. Adrisk [37] applies static analysis on top 100 commonly used ad libraries, and shows that most ad libraries collect private information, including users' location, call logs, phone number, browser bookmarks, and the list of installed applications on the phone. Moreover, some libraries directly fetch and run code from the Internet. Book et al. [11] make a longitudinal analysis of permissions used by ad libraries, and discovers that dangerous permission usage by ad libraries are increasing over time.

4.7 Conclusion

In this paper, we perform extensive empirical analysis on bad applications that are reported and removed from Google Play store. These bad applications are diligently collected by crawling Reddit forum posts and Google Play store over a three-month period. Our analysis of the data set provides a comprehensive overview of reported bad applications and their policy-violating behaviors. Our findings show that detecting copy-right violating applications and ad-aggressive applications is important for maintaining good quality of future mobile application market. Thus, we urge industry and research communities to give more attention to these areas. Our paper also includes detection of bad applications using machine learning classifiers. We derive features based on the results of our findings as well as behavior-based features. Although our current solution is performing well for detecting policy-violating applications, we believe that better solutions can be invented by focusing on each type of policy-violating applications.

Chapter 5

Detecting Camouflaged Applications On Mobile Application Markets

5.1 Introduction

With the growing number of third-party applications on mobile market places, it becomes increasingly hard to manage these applications and ensure that they are authentic, secure and of high quality. One of the emerging problems that the market owners encounter is plagiarism or cloning of mobile applications. During cloning, malicious parties copy all or parts of original applications and create similar applications or the clones. Such application plagiarism causes two main problems in mobile application markets. Firstly, it allows malicious parties to siphon revenues from original developers by replacing the advertisement libraries of plagiarised applications or by selling the clones with different prices to users. It has been shown that original developers, who are the victims of plagiarism, lost 14% of their advertising revenues and 10% of their user base to the attackers [34]. Secondly, there are cases, where attackers add malicious payloads to the clones of popular applications and threaten the security and privacy of mobile application users. In a recent study by Zheng M. et. al [104], cloning is even regarded as one of the main distribution channels of mobile malwares.

Thus, to hinder application plagiarism, a number of clone detection methods have been proposed in [101] [40] [52] [16]. However, these methods only focuses on repackaged applications, which are the clones created from the reverse-engineered codes of original applications. As such, these methods only search for code similarities among applications, consequently missing out a different set of clones, called camouflaged applications. Hence, in this chapter, we introduce the concept of camouflaged applications. Camouflaged applications are applications whose external information, such as application names, icons, user interfaces or application descriptions, are cloned. These clones may or may not have similar codes as original applications but like other clones, they plagiarise and take advantage of other applications without consensus from original developers. They are not only confusing and harmful to the users but also discourage application development by affecting developers' reputation and monetary profits.

Therefore, in this chapter, we propose a detection framework for finding camouflaged applications. Our method is based on external features of applications and applies text similarity and image similarity measurements, calculated by information retrieval systems. Although information retrieval systems have been applied to detect phishing web pages, we are the first to apply these technologies to efficiently detect camouflaged applications in mobile platforms. Our detection framework is tested with 30,625 Android applications from Google Play market. The experiment shows that 477 applications (1.56%) are potential camouflaged applications. We further analyze the behaviors of detected camouflaged applications and inspect the false alarms rate of our detection method. A total of 44 false positives, which is 9.22% of tested application samples, are identified.

This chapter is organized as follows. Problem definition of camouflaged applications and threat model are provided in Section 5.2. Background information about information retrieval systems and repackaged applications are given in Section 5.3. Our detection framework is proposed in Section 5.4 and our experiment results are shown in Section 5.5. Discussion about our findings, limitations of our method and future direction are provided in Section 5.6. After that, related work on repackaged applications are summarized in Section 5.7 and we conclude the chapter in Section 5.8.

5.2 Problem Definition

Informally, camouflaged applications are defined as "copycat" applications or "confusingly similar" applications. There have been a lot of such applications on both official Google Play store and Apple's iTunes store. Generally, the features being cloned in camouflaged applications are icons, names, screenshots and descriptions. For instance, there are camouflaged applications with very similar names, such as "Irate Birds" for the official "Angry Birds" and "Snip the Rope" for the official "Cute the Rope" [1]. Moreover, some camouflaged applications focus on screenshots to deceive users. For example, fake Pokemon Yellow application used Nintendo's popular Game Boy RPG as its application screenshots. It even managed to rise to top 3 position on iTunes store before being removed [71].

Camouflaged applications may exist on different application markets of the same platforms or across different platforms. According to Zhou et al. [101], 5-13% of the applications from unofficial Android market places are cloned from the official Google Play market. In addition, some clones may also spans across different platforms, such as Android or iOS. For instance, fake versions of popular iOS applications, such as Infinity Blade II [2] and Temple Run [3], appeared on Google Play, even before their official releases in Android version.

Market owners have imposed various developer policies for trademarks, copyrights, and patents of applications. For instance, Google Play has a policy for impersonation, stating (1) not to pretend as another company, (2) not to link to another website to represent itself as another application and (3) not to use another application's branding in title and description [73]. Moreover, Google Play's Trademark Infringement policy suggests to use distinct name, icon and logo and not to use those that are "confusingly similar" to another company's trademark. However, according to Liebergeld et al. [53], there is insufficient market control in Google Play market, because uploaded applications are not checked upfront on whether they indeed follow the policies. The policy enforcement relies heavily on feedbacks from users and developers.

[1]http://arstechnica.com/gaming/2012/08/google-play-cracks-down-on-confusingly-similar-apps/

[2]http://www.pocketgamer.co.uk/r/Android/Infinity+Blade+II/news.asp?c=43572

[3]http://m.androidcentral.com/temple-run-android-still-isnt-out-anything-else-just-malware

Threat Model: The main goal of attackers is to trick users into installing their camouflaged applications. There are two ways by which users can install applications on their mobile devices. One way is to use default installer applications, such as Play Store or iTune Store, on mobile devices. Another way is to use desktop browsers, download applications from the providers' websites and later synchronize the applications to their mobile devices. In both cases, there are two situations in which user can be tricked to install the camouflaged applications. One is during the search and another is after the user goes to the detailed information page.

- When browsing applications or searching for an application, users can only observe application icons, application names and publishing company names. Some users download applications directly from the search results, instead of going to the detailed pages. Therefore, these three pieces of information play an important role in tricking the users. Although the ranking algorithm used by the Google also plays a role, it is out of scope of this work.
- In the detailed information pages, application descriptions and screenshots are the main visual elements for users. Thus, they also play a critical role in tricking the users by attackers.

There are several ways in which attackers can gain profit for creating camouflaged applications. Table 5.1 summarizes different attackers' motivations as well as various possible attacks from camouflaged applications. Attack type may vary from mild copy-right violation and information theft to severe phishing and malware attacks. From the table, we can see that in addition to users and developers, other third-parties, such as banks and telecom providers, can be adversely affected by camouflaged applications.

5.3 Background

5.3.1 Information Retrieval Systems

Information retrieval systems are used for retrieving relevant information from a collection of information resources. Most information retrieval systems includes two processes: in-

Attacker Motivation	**Attack Type**	**Mainly Affected Parties**
Replacing advertise libraries	Copy-right violation	Developers
Creating paid version of free applications	Copy-right violation	Developers
Selling users' information to third parties	Information theft	Users
Stealing users' bank credentials	Phishing	Users and banks
Sending premium SMSes	Malware	Users and telecom providers

Table 5.1: Categorizing Attacks of Camouflaged Applications

dexing and retrieving. During indexing, the systems process documents that are either text documents or image, and extract useful information from them. During retrieving steps, query objects that are also processed, cleaned and their useful information are extracted. Then, similarity distance are measured between the query document and a collection of documents by using their representations. Ranked or sorted results are then returned to the users, together with the similarity scores.

Information retrieval techniques have also been used to detect phishing websites [95] [89]. However, the traditional phishing detection methods cannot be applied directly on platform providers' websites, such as Google Play Store. This is because camouflaged applications and original applications can be featured on the same official website. Thus, meta-data analysis of web contents, such as hyper-links, web titles, web links, etc, cannot be applied in detecting camouflaged applications.

5.3.2 Repackaging and Code-Based Detectors

Cloned applications are often the result of repackaging, which includes recovering source codes of original applications and illegally re-compiling them with different developers' certificates. Repackaging is common in Android application platform. In Android applications, Java source code are compiled into the Dalvik executable (DEX) format and run in Dalvik virtual machines. Dalvik byte codes can be easily reverse engineered by publicly available online tools, such as dex2jar and jd-gui.

As the repackaged clones are created from source codes of original applications, their source codes are similar to certain extent. Thus, code-based detectors can be used to detect repackaged applications. Generally, there are three types of code similarity detectors: feature-based, structure-based and PDG (Program Dependence Graph)-based. Feature-based detectors extracts features, such as number or size of classes, methods, loops, variables, from the applications and detects their similarities. Structure-based detectors convert applications into a stream of tokens and compare their streams. On the other hand, PDG-based detectors construct PDGs from the applications and compare them to derive the similarity scores. Many other code-based detectors, that have been proposed for repackaged applications, will be discussed more in Section 4.6.

5.4 A Framework for Detecting Camouflaged Applications

Accuracy and scalability are the key factors, considering the number of third-party applications in mobile markets. Thus, the goal of this work is to have a lightweight simple detection system, which can efficiently detects the camouflaged applications. The implementation of our framework should allow developers to check their applications before submitting to the application stores. It can also used by Google Play for vetting before or after the application submission.

Our system leverages on the light-weight information retrieval systems, such as text retrieval and content-based image retrieval systems. There are four features with which we try to find camouflaged applications: application name, description, icon and screenshot. Application name and descriptions are handled by text retrieval systems, while application icon and screenshots are handled by image retrieval system. Figure 5.1 shows the architecture of our detection system. Our detection system includes four main steps: crawling, indexing, querying and detecting.

5.4.1 Crawling

First, we need a collection of existing applications, with which the potential camouflaged applications are compared. This application collection can be from different markets of different mobile platforms, depending on where we want to detect camouflaged applica-

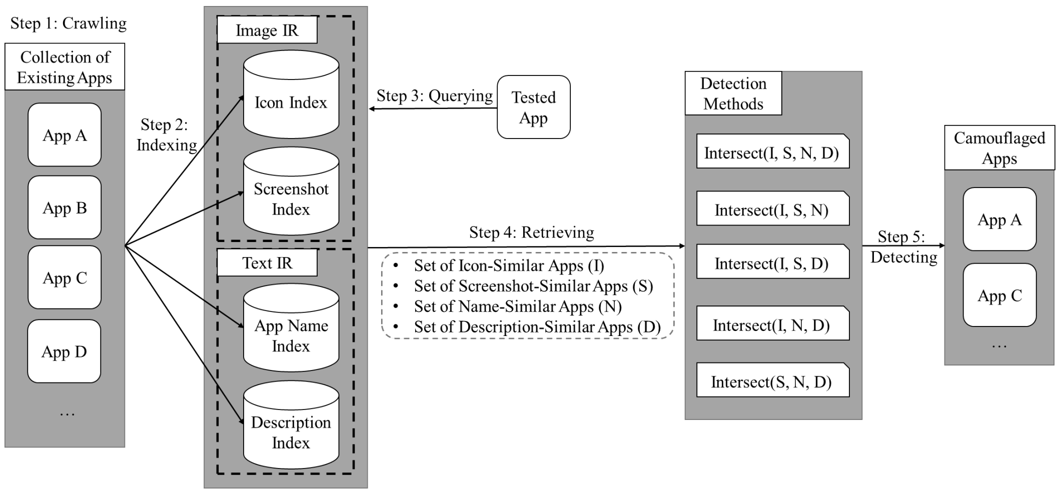


Figure 5.1: Framework for Detecting Camouflaged Applications

tions. For instance, if we want to detect camouflaged applications, which are uploaded on unofficial Android markets, existing application collection should be crawled from official Google Play market and tested applications should be crawled from unofficial Android markets. However, if we want to detect camouflaged applications on Google Play's Android market, which are copied from iTunes market, the existing application collection should be from official Google Play market and tested applications should be from Apple's iTunes market.

Our framework is independent of mobile platform and application market. It can be used on any platforms or markets as long as the market displays application names, icons, screenshots and descriptions. In our experiment, we crawled applications from official Android market and detect camouflaged applications within the same market. We use unofficial Google API to crawl App info, such as id, name, developer, rating as well as application description, icon and screenshots. Total of 30,625 applications are crawled for the experiment.

5.4.2 Indexing

The second step of most information retrieval systems is indexing. During indexing, a collection of documents are cleaned and processed to get ready for queries. We call both texts and images as "documents". Indexing can be done offline and just one time. Therefore, it is suitable for a large collection of documents. For each application in our 30,625 crawled applications, we create a name index, description index, icon index and screenshot index. Name and description indexes are created by text retrieval engines, while icon and screenshot indexes are handled by image retrieval engine.

Text Indexing

There are many types of text retrieval systems, such as boolean model, vector space model, probabilistic models. Most of them can be plugged and played in our detection framework. However, in our experiment, vector space model is used as it applies similar-word matching instead of exact-word matching algorithm. In the vector space model, each document is represented by a weighted vector in high-dimensional space. The weights from vectors are measured by TF-IDF scheme, which stands for Term Frequency (TF) and Inverse Document Frequency (IDF). Open-source software, such as Lucene [60], can be used to implement TF-IDF scheme. Tokenizing, stemming and removal of stop words are all handled by Lucene.

Image Indexing

Similar to text retrieval methods, there are also many types of image retrieval methods. They extract visual features from the images and index those features with a pointer to the parent image. The extracted features include colors, color distributions, textures or joint histograms, which involve both color and texture information. Different algorithms have their own advantages and disadvantages on performance and robustness depending on the applied scenarios and types of images. We choose auto color correlogram algorithm [44], which uses the spatial correlation of colors. The algorithm is tested using SIMPLIcity data set [85] and is shown to be both effective and inexpensive in general purpose situations [59]. Note that our framework can also be easily modified to use other visual information retrieval algorithms. We use an open-source software, LIRE [56], to perform the visual information retrieval.

5.4.3 Querying and Retrieving

The third step is to query the index databases with potential camouflaged applications. In our case, the same 30,625 crawled applications are used as potential camouflaged applications. For each queried application, we retrieved applications, which have similar user interfaces but are from different developers. Information retrieval systems are used to calculate the similarity scores, and developer ID information, obtained from Google Play website, is used to ensure that similar applications are not from the same developer.

For each query, information retrieval systems calculate the cosine similarity score between query document and a set of indexed documents. The cosine similarity score measures the similarity distance between two vector representations of documents. The score ranges from 0 to 1, where similarity score of 0 represents two totally different documents and similarity score of 1 represents two totally similar documents. The retrieved similarity score are then used to rank the documents. In our case, retrieved set of applications is sorted based on the decreasing similarity scores, meaning the most similar ones are on the top of the list. We only use top-ten similar applications in each retrieved set to reduce false positives.

The output of each queried application is four sets of similar applications, namely I, S, N and D, where

- I is a set of applications that have similar icons as queried application,
- S is a set of applications that have similar screenshots as queried application,
- N is a set of applications that have similar names as queried application and
- D is a set of applications that have similar description as queried application.

Each set contains at most ten similar applications and many sets have fewer than ten applications. Note that although we use the same application set for indexing and querying, different application set can also be applied in our architecture if we want to differentiate camouflaged applications across different markets.

5.4.4 Detecting

The fourth step of our framework is detection. Our detection method is different intersection sets of the four retrieved set I, S, N and D. This step generates the following five different result sets for each potential camouflaged application.

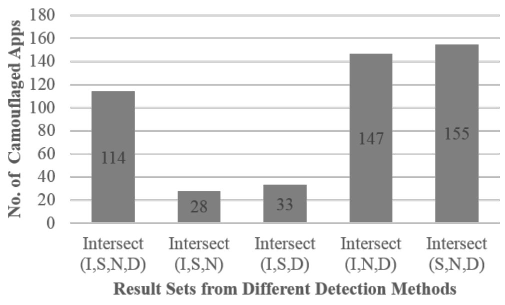


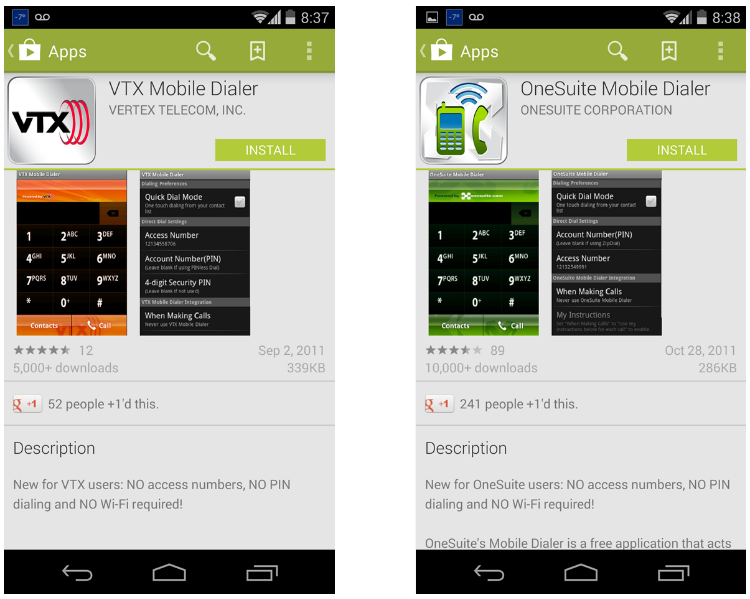
Figure 5.2: Number of Camouflaged Applications for Each Detection Method

- Intersect(I,S,N,D) is a set of applications that have similar icons, screenshots, names and descriptions as queried application,
- Intersect(I,S,N) is a set of applications that have similar icons, screenshots and names as queried application but are not included in Intersect(I,S,N,D),
- Intersect(I,S,D) is a set of applications that have similar icons, screenshots and descriptions as queried application but are not included in Intersect(I,S,N,D),
- Intersect(I,N,D) is a set of applications that have similar icons, names and descriptions as queried application but are not included in Intersect(I,S,N,D),
- Intersect(S,N,D) is a set of applications that have similar screenshots, names and descriptions as queried application but are not included in Intersect(I,S,N,D).

Since these sets contain very similar applications from different developers, they are considered as camouflaged applications. Nonetheless, there can also be false alarms, where the result set contains non-camouflaged applications. False alarms are created because information retrieval methods cannot differentiate them, although they are obvious to normal users that they are not camouflaged applications.

5.5 Experiment and Results

Out of 30,625 applications, we find that 477 applications (1.56%) have 1 to 6 camouflaged applications. Figure 5.2 shows the exact number of camouflaged application from each result set. According to the figure, we can see that Intersect(I,S,N,D), Intersect(I,N,D)



New for VTX users: NO access numbers, NO PIN dialing and NO Wi-Fi required!

New for OneSuite users: NO access numbers, NO PIN dialing and NO Wi-Fi required!

OneSuite's Mobile Dialer is a free application that acts

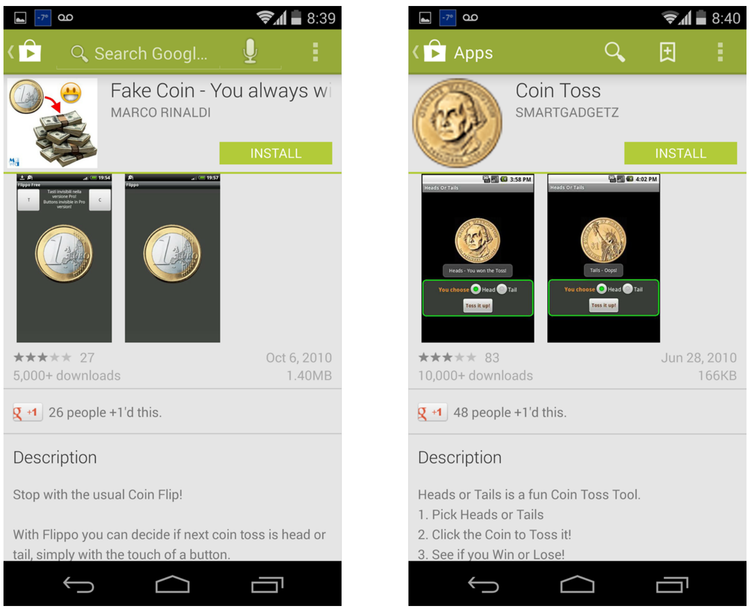
Figure 5.3: Example of Detected Camouflaged Application

and Intersect(S,N,D) reports more camouflaged applications than Intersect(I,S,N) and Intersect(I,S,D) methods.

An example of detected camouflaged applications, namely "VTX Mobile Dialer" and "OneSuite Mobile Dialer", is shown in Figure 5.3. The two applications have the very similar screenshots, application name and description. Thus, they are reported in Intersect(S,N,D) set. However, they use different developer IDes as well as different contact information. The developer website and email address of "VTX Mobile Dialer" are https://www.vtxtelecom.com/ and *mobileapp@vtxtelecom.com*. On the other hand, the developer website and email address of "VTX Mobile Dialer" are http://www.onesuite.com/ and *mobileapp@onesuite.com*. Although they claim to be from different companies, their user interfaces are suspiciously similar. Therefore, they are regarded as camouflaged applications.

Determining the False Alarms: Determining the false positives and false negatives for camouflaged applications is a challenge, as we do not have any ground truth samples. Thus, we decide to do manual inspection on the result sets to determine the false positives. Though tedious, expert manual inspection has been a common way to test the efficiencies of information retrieval systems. To our surprise, a lot of the reported camouflaged applications have almost identical user interfaces. This makes our manual inspection easier.

Our manual inspection shows that the result sets contain a total of 44 false positives,



Fake Coin - You always w
MARCO RINALDI
27
Oct 6, 2010
5,000+ downloads
1.40MB
26 people +1'd this.
Description
Stop with the usual Coin Flip!
With Flippo you can decide if next coin toss is head or tail, simply with the touch of a button.

Coin Toss
SMARTGADGETZ
83
Jun 28, 2010
10,000+ downloads
166KB
48 people +1'd this.
Description
Heads or Tails is a fun Coin Toss Tool.
1. Pick Heads or Tails
2. Click the Coin to Toss it!
3. See if you Win or Lose!

Figure 5.4: Example of False-Positive Camouflaged Applications

which is 9.22% of reported camouflaged applications. However, false positives exist only in the Intersect(I,N,D) and Intersect(S,N,D). Intersect(I,N,D) contains 21 false positive samples and Intersect(S,N,D) contains 23 false positive samples. No false positive applications have been identified in Intersect(I,S,N,D), Intersect(I,S,N) and Intersect(I,S,D), which consider the similarity of both icons and screeenshots. This indicates that icons and screenshot similarity measures are great indicators of camouflaged applications.

Figure 5.4 shows an example of false alarm applications, called "Fake Coin - You always win!" and "Coin Toss". Although their user interfaces are similar, it is quite obvious to the real users that they provide different functions: the former application is for tricking friends and the latter application is for randomly tossing the coin. Therefore, these two applications should not be regarded as camouflaged applications.

5.6 Discussion

In this section, we will discuss about our findings on camouflaged applications as well as limitations of our method and future work.

Feature Selection in Detection Method: Our detection method is limited to camouflaged applications with at least three similar features. Nonetheless, there can still be camouflaged applications with only one or two similar features. For instance, there are camouflaged applications with only similar icons. Although our method can be easily extended to find applications with one or two similar features, many applications use very simple and easily searchable icons, such as a light bulb. Consequently, there are a lot of false alarms when we use only two features. Thus, it is still a challenge on how to ensure quality control on icons and names of applications in the market.

Applications from Open-Source Projects: Our result shows that there are applications, which are modified from open-source projects, such as e-book readers, music players and map applications. Although they use different contents, such different books or songs, and change the themes, the applications are still highly similar as they use source codes from the same projects. Thus, although they do not copy from each other, they are still considered as camouflaged applications in our framework.

Applications with Different Versions: We find out that some camouflaged applications claim to be different versions from one another. They use version differentiating words, such as “HD” (High Definition), “full”, “II” (two), “plus” and “pro”. However, many of them do not provide additional functionalities, although they claim to be upgraded versions. It is possible that a malicious attacker tries to attract more customers by claiming to provide upgraded version of the victim application. To solve this problem, application markets should enforce that developers use the same account, when they claim to provide upgraded version of an existing application. Our detection framework for camouflaged applications can serve as an automatic policy enforcement mechanism for these kind of applications.

Internationalized Applications: Another finding of our experiment is that many international companies, such as banks, have different applications developed for different countries and languages. Unfortunately, they also use different developer ID in Google Play to update them. For instance, “Banco Weng Hang, S.A.” application uploaded by “Banco Weng Hang, S.A.” provides banking services in Chinese, while “Wing Hang Bank” application uploaded by “Wing Hang Bank Ltd” provides the same services in English. This is actually a vulnerability, which allows attackers to impersonate as legitimate applications and launch phishing attacks.

5.7 Related Work

Studies on the repackaged applications have become popular recently. Zhou et. al. [104] studies 1260 Android malwares and finds out that 1083 malwares are repackaged applications. Balanza et al.[8] analyzes a repackaged malware, called DroidDreamLight and states that trojanizing or repackaging is common form of infection in Android market. Jung et al. [47] launches repackaging attack on bank applications. Moreover, Vidas et al. [83] shows that some malwares are even repackaged with the valid certificates from original developers. It also proposes an authentication protocol for market applications which makes it difficult for an attacker to perform repackaging.

Chen et al. [13] also studies the underground economy of Android application plagiarism. Similarly, Gibler et al. [34] studies the impact of repackaged applications and finds out that 14% of original developers' revenues and 10% of user based are redirected to the attacker. Zheng et al. [97] presents various obfuscation techniques which allow automatic repackaging of original malwares to different variants. Transformed malwares are then used to test the robustness of Android anti-virus systems. Potharaju et al. [74] uses permission information and estimates that 29.4% of applications are likely to be plagiarized. They also detect repackaged applications using Deckard [46], which is a tree-based detection algorithm of cloned codes.

DroidMOSS [101] and Juxtapp [40] [52] and apply fuzzy hashing on program instruction sequence and derive the similarity score by calculating the edit distance between two generated fingerprints. Crussell et al. [16] proposes DNADroid, which uses Program Dependence Graph(PDG) to determine code similarity. DNADroid is similar to our approach because it filters the applications based on application names, packages, markets, owners and descriptions. However, such filtering is performed only to make the PDG comparison more scalable for determining the similarity between two applications.

AnDarwin [17] applies Locality-Sensitive Hashing (LSH) to detect the repackaged applications. Zhou et al. [100] calls repackaged applications as "piggybacked" applications and proposes linearithmic search algorithm in a metric space to detect them. Desnos et al. [22] proposes an algorithm, which uses Normalized Compression Distance (NCD) to analyze the similarity and differences between two Android applications. Similarly, Lin et al. [54] apply thread-grained system call sequences to detect repackaged applications. Ko et al. [49] extract k-gram based software birthmarks from the dissembled codes and measure the similarity of DEX files.

Huang et al. [43] proposes an evaluation framework for detection algorithms of repackaged application by measuring their resilience to obfuscation methods. Different from other approaches, [99] proposes to use software watermarking to prevent repackaging. In summary, researchers have proposed different ways of detecting repackaged applications by measuring the source code similarity or software watermarking. However, none of them have yet considered camouflaged applications, which have very similar user interfaces, instead of similar source codes.

5.8 Conclusion

This chapter highlights the existence of camouflaged applications in mobile application markets as well as their exposed risk on application users and developers. Although there have been papers about repackaged applications and their copy-right infringement, our work is the first to introduce the concept of camouflaged applications and consider their user interface similarity. Our work describes a proper threat model of camouflaged applications, including their attack scenarios and attackers' motivations. Moreover, we propose a simple, yet effective, detection framework, which applies text and image retrieval systems that are accurate and scalable in detecting camouflaged applications. The proposed framework is tested and the experiment result shows that 477 applications are camouflaged. We analyze these camouflaged applications, discuss their behaviors and calculate the false alarm rates. Our work shows that detecting camouflaged applications is important, not only for maintaining a safe mobile application market but also for controlling the quality of mobile applications.

Chapter 6

Dissertation Conclusion and Future Work

6.1 Summary of Contribution

This dissertation makes contributions on vulnerability analysis of mobile application frameworks and detection mechanisms on policy-violating applications.

Our first work focuses on finding vulnerabilities and launching attacks on iOS framework. We proposes generic attack vectors that can be launched on iOS by an approved third-party application. During attacks, an attacker bypasses vetting process and sandbox mechanisms by dynamically loading frameworks and invoking C functions. We provide several proof-of-concept attacks using these attack vectors. The results of our first work have been reported to Apple's iOS security team. They have been acknowledged by Apple and included in news media. The work has also published in the 11th International Conference on Applied Cryptography and Network Security (ACNS 2013).

Our second work proposes to find vulnerabilities in Android framework by uncovering unprotected APIs. Several analysis, including call graph analysis, component analysis and data flow analysis, are performed to retrieve unprotected APIs. After that, we launched several proof-of-concepts attacks on Android devices. Our findings have been reported to Google and Google has publicly acknowledged our contribution in its Security Bulletin. This work has been published in the 14th International Conference on Privacy, Security and

Trust (PST 2016).

Our third work performs empirical analysis on policy-violating applications. This is the first time in literature to analyze the categories and behaviors of real-life samples of policy-violating applications. We also propose several features that can be used to detect policy-violating applications with machine learning algorithms. This work has been published in the 11th International Conference on Malicious and Unwanted Software (Malcon 2016). It is also filed for patent with Hua Wei at the time when this dissertation was submitted.

Our fourth work detects camouflaged applications, which violate intellectual property policies of application markets. Text similarity and image similarity scores are used to detect these applications. The work is published in the 17th Annual International Conference on Information Security and Cryptology (ICISC-2014).

This dissertation shows several weaknesses in the security of mobile frameworks, and proposes various ways of improvements in ensuring security and privacy of mobile users.

6.2 Future Direction

We identify the following future directions:

6.2.1 Improvement on Current Vulnerability Analysis

An interesting way to extend our analysis is to assign a risk level to each vulnerability discovered. Clearly, not every unprotected API poses a high risk. The risk level may be calculated based on the nature of an exposed API, such as whether it accesses Kernel files. We may also use heuristics in categorizing vulnerable APIs. For example, if an exposed API contains source codes related to accessing Content Provider and returning results, it should be categorized as potential privacy leakage. More effort in this aspect should be made so as to develop an industry standard for evaluating and protecting mobile framework APIs.

6.2.2 Vulnerability Analysis on Other Frameworks

One of our future directions in this field is to discover framework-level vulnerabilities in smart watches, smart TVs and smart cars, since they are also based on Android mobile systems, such as Android Auto and Android Wear. IOT platforms, such as Brillo, are also interesting to explore for security vulnerabilities. When new systems are added into mobile infrastructure, it make it more difficult and complex to preserve the security and privacy of mobile users. Currently, there are no standard vulnerability analysis tools specifically designed for these new systems, and creating these tools will be extremely helpful in ensuring security and privacy of mobile users. ach vulnerability analysis will be tailored towards specific systems, since their security mechanisms are different from others.

6.2.3 Vulnerability Analysis on Third-Party APIs

Another interesting future direction is to extend our study to third-party applications. Like system services, some third-party applications provide AIDL interfaces to other third-party applications. For example, many APIs are provided by Google Play services, including Google Map, Google Plus, In-App Billing, and Google Wallet. Although such services are not part of Android platform, they are supported by most Android mobile devices. Another example is Dropbox, which allows other applications to access and manipulate a user's Dropbox account. It remains interesting to investigate the unprotected APIs of the AIDL interfaces provided by vendor applications, such as Google Play services and other third-party applications, such as Dropbox.

Bibliography

[1] Y. Aafer, W. Du, and H. Yin. *Security and Privacy in Communication Networks: 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, chapter DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android, pages 86–103. Springer International Publishing, Cham, 2013.

[2] U. M. André Egners, Björn Marschollek. Hackers in your pocket: A survey of smartphone security across platforms. Number AIB-2012-07, may 2012.

[3] Android. Enforcing permissions in androidmanifest.xml. http://developer.android.com/guide/topics/security/permissions.html. [Online; accessed 9-October-2014].

[4] Apple. About the security content of ios 7. https://support.apple.com/en-sg/HT202816. Accessed: 2016-07-01.

[5] apple.com. Apple open source projects. http://www.apple.com/opensource/. Accessed: 2013-01-01.

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[7] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Enabling generic and extensible access control on android. *CoRR*, abs/1404.1395, 2014.

[8] M. Balanza, O. Abendan, K. Alintanahin, J. Dizon, and B. Caraig. Droiddreamlight lurks behind legitimate android apps. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, pages 73–78, Washington, DC, USA, 2011. IEEE Computer Society.

[9] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 274–277, New York, NY, USA, 2012. ACM.

[10] M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 96–111, Washington, DC, USA, 2011. IEEE Computer Society.

[11] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. *CoRR*, abs/1303.0857, 2013.

[12] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: Analyzing android applications for capability leak. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 125–136, New York, NY, USA, 2012. ACM.

[13] H. Chen. Underground economy of android application plagiarism. In *Proceedings of the First International Workshop on Security in Embedded Systems and Smartphones*, SESP '13, pages 1–2, New York, NY, USA, 2013. ACM.

[14] N. Chen, S. C. Hoi, S. Li, and X. Xiao. Simapp: A framework for detecting similar mobile applications by online kernel learning. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 305–314, New York, NY, USA, 2015. ACM.

[15] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[16] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proceedings of 17th European Symposium on Research in Computer Security*, pages 37–54, 2012.

[17] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar android applications. In *18th European Symposium on Research in Computer Security*, ESORICS '13, Egham, U.K., 2013.

[18] D. Damopoulos, G. Kambourakis, and S. Gritzalis. isam: An iphone stealth airborne malware. In J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, editors, *SEC*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 17–28. Springer, 2011.

[19] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] K. DAVID. Jailbreaking iphone legal, u.s. government says. http://abcnews.go.com/Technology/story?id=11254253. Accessed: 2013-01-01.

[21] A. Desnos. Androguard, 2011.

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

[23] A. Developer. Xcode, apples integrated development environment for creating apps for mac and ios. https://developer.apple.com/xcode/. Accessed: 2013-01-01.

[24] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*. The Internet Society, 2011.

[25] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

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

[27] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[28] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[29] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. 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 '11, pages 3–14, New York, NY, USA, 2011. ACM.

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

[31] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission redelegation: Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[32] J. Freeman. Cydia, an alternative to apples app store for jailbroken ios devices. http://http://cydia.saurik.com/. Accessed: 2013-01-01.

[33] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 431–444, New York, NY, USA, 2013. ACM.

[34] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. *Proceedings of 11th International Conference on Mobile Systems, Applications and Services*, 2013.

[35] Google. Nexus security bulletin - march 2016. https://source.android.com/security/bulletin/2016-03-01.html. Accessed: 2016-07-01.

[36] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[37] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM.

[38] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[39] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.

[40] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *Proceedings of the 9th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 62–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[41] T. Hao, G. Xing, and G. Zhou. isleep: Unobtrusive sleep quality monitoring using smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 4:1–4:14, New York, NY, USA, 2013. ACM.

[42] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1005–1019, San Diego, CA, Aug. 2014. USENIX Association.

[43] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In M. Huth, N. Asokan, S. apkun, I. Flechais, and L. Coles-Kemp, editors, *Trust and Trustworthy Computing*, volume 7904 of *Lecture Notes in Computer Science*, pages 169–186. Springer Berlin Heidelberg, 2013.

[44] J. Huang, S. R. Kumar, M. Mitra, W.-J. Zhu, and R. Zabih. Image indexing using color correlograms. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, CVPR '97, pages 762–, Washington, DC, USA, 1997. IEEE Computer Society.

[45] A. P. Info. App store tops 40 billion downloads with almost half in 2012. http://www.apple.com/pr/library/2013/01/07App-Store-Tops-40-Billion-Downloads-with-Almost-Half-in-2012.html. Accessed: 2013-01-01.

[46] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[47] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi. Repackaging attack on android banking applications and its countermeasures. *Wirel. Pers. Commun.*, 73(4):1421–1437, Dec. 2013.

[48] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[49] J. Ko, H. Shim, D. Kim, Y.-S. Jeong, S.-j. Cho, M. Park, S. Han, and S. B. Kim. Measuring similarity of android applications via reversing and k-gram birthmarking. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, pages 336–341, New York, NY, USA, 2013. ACM.

[50] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[51] L. Li, A. Bartel, J. Klein, and Y. Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*. IEEE, 2014.

[52] S. Li. Juxtapp and dstruct: Detection of similarity among android applications. Master's thesis, EECS Department, University of California, Berkeley, May 2012.

[53] S. Liebergeld and M. Lange. Android security, pitfalls and lessons learned. In *ISCIS*, pages 409–417, 2013.

[54] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. *Identifying android malicious repackaged applications by thread-grained system call sequences*. Computers and Security, Greenwich, CT, USA, 2013.

[55] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[56] M. Lux and S. A. Chatzichristofis. Lire: lucene image retrieval: an extensible java cbir library. In *Proceedings of the 16th ACM international conference on Multimedia*, MM '08, pages 1085–1088, New York, NY, USA, 2008. ACM.

[57] macgasm.net. It professionals rank ios as most secure mobile os. http://www.macgasm.net/2012/08/17/it-professionals-rank-ios-as-most-secure-mobile-os/. Accessed: 2013-01-01.

[58] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1457–1462, New York, NY, USA, 2012. ACM.

[59] O. Marques and M. Lux. Visual information retrieval using java and lire. In W. R. Hersh, J. Callan, Y. Maarek, and M. Sanderson, editors, *SIGIR*, page 1193. ACM, 2012.

[60] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., Greenwich, CT, USA, 2010.

[61] M. Mitchell, G. Tian, and Z. Wang. Systematic audit of third-party android phones. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 175–186, New York, NY, USA, 2014. ACM.

[62] NakedSecurity. First iphone worm discovered - ikee changes wallpaper to rick astley photo. https://nakedsecurity.sophos.com/2009/11/08/iphone-worm-discovered-wallpaper-rick-astley-photo/. Accessed: 2013-01-01.

[63] NakedSecurity. Hacked iphones held hostage for 5 euros. https://nakedsecurity.sophos.com/2009/11/03/hacked-iphones-held-hostage-5-euros/. Accessed: 2013-01-01.

[64] L. T. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. S. Kim, P. Tague, and J. Zhang. Unlocin: Unauthorized location inference on smartphones without being caught. In *2013 International Conference on Privacy and Security in Mobile Systems, PRISMS 2013, Atlantic City, NJ, USA, June 24-27, 2013*, pages 1–8, 2013.

[65] S. Nicolas. ios 6 runtime headers. https://github.com/nst/iOS-Runtime-Headers. Accessed: 2013-01-01.

[66] S. Nicolas. iphone privacy. Black Hat DC (2011). Accessed: 2013-01-01.

[67] S. Nicolas. Objective-c runtime browser, for mac os x and ios. https://github.com/nst/RuntimeBrowser/. Accessed: 2013-01-01.

[68] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[69] J.-S. Oh, M.-W. Park, and T.-M. Chung. The solution of denial of service attack on ordered broadcast intent. In *Advanced Communication Technology (ICACT), 2014 16th International Conference on*, pages 397–400, Feb 2014.

[70] T. Online. Local researchers help fix ios security flaws. http://www.todayonline.com/tech/local-researchers-help-fix-ios-security-flaws. Accessed: 2016-07-01.

[71] K. Orland. Fake pokemon yellow rises to no. 3 position on itunes app charts, 2012.

[72] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and api calls. In *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, ICTAI '13, pages 300–305, Washington, DC, USA, 2013. IEEE Computer Society.

[73] G. Play. Intellectual property.

[74] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: attack strategies and defense techniques. In *Proceedings of the 4th international conference on Engineering Secure Software and Systems*, ESSoS'12, pages 106–120, Berlin, Heidelberg, 2012. Springer-Verlag.

[75] Safe and Savvy. How secure is your iphone. http://safeandsavvy.f-secure.com/2012/06/29/how-secure-is-your-iphone/. Accessed: 2013-01-01.

[76] R. Schlegel, K. Zhang, X. yong Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society, 2011.

[77] N. Security. Apple lets malware into app store. https://nakedsecurity.sophos.com/2011/11/08/apples-app-store-security-compromised/. Accessed: 2013-01-01.

[78] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, February 2016.

[79] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang. Detecting clones in android applications through analyzing user interfaces. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 163–173, Piscataway, NJ, USA, 2015. IEEE Press.

[80] S. Times. Apple fixes ios 7 after singapore researchers identify flaws. http://www.straitstimes.com/singapore/apple-fixes-ios-7-after-singapore-researchers-identify-flaws.

[81] TrendLabs. Malware for ios? not really. http://blog.trendmicro.com/trendlabs-security-intelligence/malware-for-ios-not-really/. Accessed: 2013-01-01.

[82] O. Tripp and J. Rubin. A bayesian approach to privacy enforcement in smartphones. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 175–190, San Diego, CA, Aug. 2014. USENIX Association.

[83] T. Vidas and N. Christin. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Proceedings of the third ACM conference on Data and application security and privacy*, CODASPY '13, pages 197–208, New York, NY, USA, 2013. ACM.

[84] VirusTotal. Virustotal public api v2.0.

[85] J. Z. Wang, J. Li, and G. Wiederhold. Simplicity: Semantics-sensitive integrated matching for picture libraries. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(9):947–963, Sept. 2001.

[86] D. Wu, X. Luo, and R. K. Chang. A sink-driven approach to detecting exposed component vulnerabilities in android apps. *arXiv preprint arXiv:1405.6282*, 2014.

[87] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[88] N. Xia, H. H. Song, Y. Liao, M. Iliofotou, A. Nucci, Z.-L. Zhang, and A. Kuzmanovic. Mosaic: Quantifying privacy leakage in mobile networks. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 279–290, New York, NY, USA, 2013. ACM.

[89] G. Xiang and J. I. Hong. A hybrid phish detection approach by identity discovery and keywords retrieval. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 571–580, New York, NY, USA, 2009. ACM.

[90] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. Intentfuzzer: Detecting capability leaks of android applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 531–536, New York, NY, USA, 2014. ACM.

[91] Z. Yang and M. Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*, WCSE '12, pages 101–104, Washington, DC, USA, 2012. IEEE Computer Society.

[92] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 1043–1054, New York, NY, USA, 2013. ACM.

[93] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing and Multimedia*, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013. ACM.

[94] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[95] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 639–648, New York, NY, USA, 2007. ACM.

[96] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser. *Data and Applications Security and Privacy XXVIII: 28th Annual IFIP WG 11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings*, chapter FSquaDRA: Fast Detection of Repackaged Applications, pages 130–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[97] M. Zheng, P. P. C. Lee, and J. C. S. Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *Proceedings of the 9th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 82–101, Berlin, Heidelberg, 2013. Springer-Verlag.

[98] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 199–210, New York, NY, USA, 2014. ACM.

[99] W. Zhou, X. Zhang, and X. Jiang. Appink: Watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

[100] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, CODASPY '13, pages 185–196, New York, NY, USA, 2013. ACM.

[101] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[102] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 409–423, Washington, DC, USA, 2014. IEEE Computer Society.

[103] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[104] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, pages 95–109. IEEE Computer Society, 2012.

[105] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. *Trust and Trustworthy Computing: 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, chapter Taming Information-Stealing Smartphone Applications (on Android), pages 93–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[106] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. *Trust and Trustworthy Computing: 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, chapter Taming Information-Stealing Smartphone Applications (on Android), pages 93–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.