DroidRay: A Security Evaluation System for Customized Android Firmwares

Min Zheng, Mingshen Sun, and John C.S. Lui
Department of Computer Science and Engineering
The Chinese University of Hong Kong

ABSTRACT

Android mobile devices are enjoying a lion's market share in smartphones and mobile devices. This also attracts malware writers to target the Android platform. Recently, we have discovered a new Android malware distribution channel: *releasing malicious firmwares with pre-installed malware to the wild*. This poses significant risk since users of mobile devices cannot change the content of the malicious firmwares. Furthermore, pre-installed applications have "*more permissions*" (i.e., silent installation) than other legitimate mobile apps, so they can download more malware or access users' confidential information.

To understand and address this new form of malware distribution channel, we design and implement "*DroidRay*": a security evaluation system for customized Android firmwares. DroidRay uses both static and dynamic analyses to evaluate the firmware security on both the application and system levels. To understand the impact of this new malware distribution channel, we analyze 250 Android firmwares and 24,009 pre-installed applications. We reveal how the malicious firmware and pre-installed malware are injected, and discovered 1,947 (8.1%) pre-installed applications have signature vulnerability and 19 (7.6%) firmwares contain pre-installed malware. In addition, 142 (56.8%) firmwares have the default signature vulnerability, five (2.0%) firmwares contain malicious *hosts* file, at most 40 (16.0%) firmwares have the native level privilege escalation vulnerability and at least 249 (99.6%) firmwares have the Java level privilege escalation vulnerability. Lastly, we investigate a real-world case of a pre-installed zero-day malware known as `CEPlugnew`, which involves 348,018 infected Android smartphones, and we show its degree and geographical penetration. This shows the significance of this new malware distribution channel, and DroidRay is an effective tool to combat this new form of malware spreading.

ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.

http://dx.doi.org/10.1145/2590296.2590313.

1. INTRODUCTION

In the past few years, we have experienced an exponential growth of mobile devices. Among the popular platforms, Android is enjoying a lion's share of mobile market. According to the recent report by the International Data Corporation (IDC) [13], Android market share in smartphone dominates nearly 80% worldwide during the second quarter of 2013. In addition, sales of smartphones grows another 32.7% in 2013, and the shipments could exceed 958.8 million units. According to VR-ZONE [30], US, UK and Japan only contributed 20.5% of the total shipments of smartphones in Q2, 2013, while the major bulk of smartphones (or 79.5%) were manufactured in countries such as China, India and others. According to the China Daily [5], shipments of smartphones on the Chinese mainland hit 18 million in April, 2012, accounting for more than half of the mobile phone market, with more than 77% of the shipment was contributed by local brands targeting mid-end and low-end customers. These statistics show that there exist a large percentage of "*low-end smartphones*" produced by low cost manufacturers in developing countries and these devices domain the Android smartphone market. Therefore, the exponential growth of Android and the abundance of low-end smartphones attract hackers to develop new ways to distribute malware.

The conventional approach to spread Android malware is by uploading the malware to Android app markets. However, with the passage of time, people pay more attention on malware detection. As of today, nearly all of the app markets scan their apps before publishing them to the public.

Recently, we have discovered that malware writers are using new channels to distribute malware: "*Malware writers pay money to manufacturers of low-cost mobile devices to pre-install malware in their devices, or they release malicious firmwares with pre-installed malware to the wild*". Note that for some manufacturers, their business line is to produce low-cost smartphones or mobile devices. To improve their sales, they usually prefer to "*pre-install*" as many applications as possible. So with the financial offer given by malware writers and their own business needs, these manufacturers have the economic incentive to include these applications (even if some of them are malware) [17] [31]. Figure 1 depicts the approach to distribute the pre-installed apps in firmwares. The manufacturer receives money (and malware) from hackers. Then the manufacturer produces and sales these smartphones to customers with these pre-installed malware in the devices. Finally, malware writers gain control of the victims' devices. As an evidence of this form of malware distribution channel, we found that jSMSHider and Oldboot are

malware family which targets custom firmwares and these familes were reported by Lookout [15] and Qihoo360 [41].

Another closely related distribution channel which we discovered was that malware writers *release firmwares of mobile devices on the Internet* in order to attract potential manufacturers or customers to download. In fact, some low-cost manufacturers often download firmwares from the wild and use them in their production. Of course, these firmwares may contain many vulnerabilities and pre-installed malware. We carried out our investigation and found that there are many customized firmwares download sites available on the Internet (e.g., [22, 32, 16, 7]).

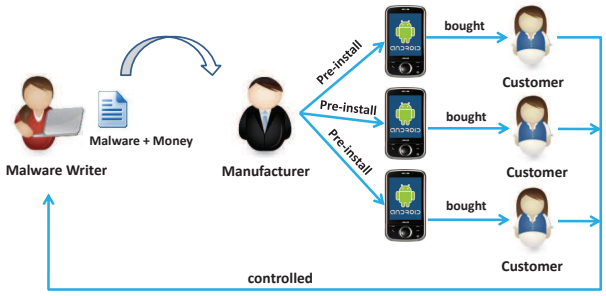

Figure 1: Malicious Android firmware distribution

The firmware contains the operating system (e.g., Android kernel) as well as some "*pre-installed*" mobile applications. Usually, users of mobile devices cannot change the content in the firmwares unless they have the root privilege or they have jailbroken the device. In addition, pre-installed applications usually have *more* permissions than other mobile applications which are downloaded from various sites (e.g., Google Play or other third party markets). Due to the enhanced permissions, pre-installed applications can use system-level APIs to perform privileged operations such as silent installation, which implies they can download more malware from the wild. In [12], authors investigate whether the pre-installed applications have *capability leak*, which means that an application can gain access to permissions without explicitly requesting them. They reported that most of these Android devices have more than 100 pre-installed applications and they pose a high risk of capability leak. However, their research only focused on the capability leak detection of pre-installed applications. It is very dangerous that if hackers release some infected firmwares in the wild, or include malware in the firmwares, it is difficult for users to determine the security level of these pre-installed applications. Further complicating the situation is that these pre-installed applications cannot be uninstalled unless users have the root privilege. In addition, not only can the hackers add malware in the firmware, but also modify the Android system. For example, the hackers can change the *iptables* to redirect users' internet visit or leave some back doors in the firmware system for the further actions.

To address the security problems of malicious firmware and pre-installed applications, we propose "*DroidRay*", a security evaluation system for customized Android firmwares. The system uses both static and dynamic analyses to evaluate the firmware security on both the application and system levels. The main contributions of this paper are:

- We design a security evaluation system for customized Android firmwares on both the application level and system level. We study how the malicious firmwares and malware are injected into the market. In particular, we use DroidRay to perform application signature vulnerability detection and malware detection on the application level (see Sec. 4). Then it performs system signature vulnerability detection, network security analysis, and privilege escalation vulnerability detection for the customized Android firmware on the system level (see Sec. 5).

- We use "*DroidRay*" to systematically analyze 250 customized Android firmwares and 24,009 pre-installed applications. On application level, we successfully discovered 1,947 (8.1%) applications have signature vulnerability and 19 (7.6%) firmwares contain pre-installed malware. On system level, we discovered 142 (56.8%) firmwares have the default signature vulnerability, five (2.0%) firmwares contain malicious *hosts* file, at most 40 (16.0%) firmwares have the native level privilege escalation vulnerability and at least 249 (99.6%) firmwares have the Java level privilege escalation vulnerability (see Sec. 4, 5).

- To show the influence of pre-installed Android malware, we investigate a real world case of a pre-installed zero-day Android malware known as `CEPlugnew`, which involves 348,018 infected Android smartphones. We investigate the degree of its penetration as well as its infected geographical regions (see Sec. 6).

The paper is organized as follows: In Section 2, we briefly overview the background on Android firmware. Then we discuss the system design and analysis methodology of DroidRay in Section 3. In Section 4, we present our systematic study of 24,009 pre-installed applications to show their security problems. In Section 5, we present our systematic study of 250 customized Android firmwares. In Section 6, we present our detailed analysis of a real world malware to illustrate the Android malware ecosystem. Related work is given in Section 7 and Section 8 concludes.

2. BRIEF OVERVIEW OF ANDROID FIRMWARE

In this paper, Android firmware refers to a "*packed binary system image*" which can be written to a non-volatile storage, say a device's flash memory. "*Flashing*" is a process of writing data to the internal memory on smartphones. Normally, one needs to flash a firmware to a smartphone in order to upgrade (or even downgrade) the system software.

There are two types of firmwares: (1) firmware with "*recovery partition*" and, (2) firmware "*without recovery partition*". The recovery partition can be considered as an alternative boot partition which lets the device boot into a recovery console for performing advanced recovery and maintenance operations. For example, it can perform an update using the "`update.zip`" file on the microSD card. There are two types of recovery partitions, one is the "*official recovery partition*", the other is the "*customized recovery partition*". The official recovery partition will check the public key information of the firmware. Only the official firmwares can be flashed into an Android device. The custom recovery

partition does not check whether the public key is official or not, so users can flash third-party firmwares into Android devices. Usually, if an Android smartphone has a recovery partition, users could just download those firmwares without recovery partition, and then perform the flashing.

All firmwares have three core components: (1) the "`system`" folder, (2) the "`META-INF`" folder and (3) the "`boot.img`" file and several optional components (e.g., the "`data`" folder). The "`system`" folder contains most of the system files, and the "`/system/app`" directory on the Android device is where the pre-installed applications with system level privilege are stored. In addition, the "`/system`" folder is read-only by default, which means user cannot uninstall the pre-installed applications unless the user has the root privilege. The "`META-INF`" folder contains the installation script and signature information of the firmware. Because firmware's developers can create their own installation scripts, they can put their applications into the "`/system/app`" folder or other folders. Malware writers who create malicious firmwares often craft a script which can copy pre-installed applications into "`/system/app`" folder or "`/data/app`" folder. Therefore, the pre-installed apps can be in any sub-directory because malware writers can inject various installation scripts. In order to bypass the malware detection, the malware writer may place the malware into another folder, or even encrypt these applications if necessary. This is the reason why some pre-installed applications are not stored in the "`/system/app`" folder of the firmware. This also points to the weakness of current Android anti-virus software since they only inspect applications in the `/system/app` folder. Last but not least, the "`boot.img`" is the Linux kernel which is used in the Android device. This `img` file also contains many initialization scripts for system booting, so the hackers can perform modification in order to hide their malicious intent.

3. SYSTEM DESIGN AND ANALYSIS METHODOLOGY

DoridRay is a security evaluation system for customized Android firmwares. The system uses Android firmwares as the input, then the system analyzes both Android firmwares as well as pre-installed applications. After the analysis, DroidRay outputs the analysis report of these firmwares and pre-installed applications. In the analysis process, DoridRay uses both static analysis and dynamic analyses to evaluate the firmware security on application level and system level. The architecture of DroidRay is depicted in Figure 2.

Let us discuss the methodology we use to carry out the security analysis of various Android firmwares. The methodology can be summarized as the following steps:

- We download and collect firmwares from different Android firmware forums and websites.
- After we obtain these firmwares, we use DroidRay to analyze the configuration files (e.g., "`/system/build.prop`" file) to obtain the firmware information. The information includes firmware name, product model and Android version, etc.
- DroidRay uses both static and dynamic methodologies to extract and analyze the pre-installed applications in the Android firmware. Specifically, we carry out two forms of security analysis: (a) application signature vulnerability detection and, (b) malware detection. The details are presented in Section 4.
- DroidRay uses both static and dynamic methodologies to analyze the system security of the Android firmware. Specifically, we carry out three forms of security analysis: (a) system signature vulnerability detection, (b) network security analysis, and (c) privilege escalation vulnerability detection. The details are presented in Section 5.
- DroidRay organizes all the analysis results of systems and pre-installed applications into the database and then outputs various reports which are classified by the signature, MD5 and package name. This helps security analysts use DroidRay to quickly identify and associate malware and vulnerabilities.

To show the effectiveness of our system, we used DroidRay to perform a large scale security analysis on 250 Android firmwares and 24,009 applications. To the best of our knowledge, this is the largest scale of Android firmware security analysis reported so far. We downloaded these Android firmwares from [22, 16, 36, 19, 29, 7, 4]. The statistics are summarized in Table 1. Because of the page limit, we put the full firmware information at `https://www.dropbox.com/s/8rrmyqwzlqg3wjw/RomInfo.xls`.

Firmware Information	Number
Total firmwares	250
Total number of apps in all firmwares	24,009
Average number of apps per firmware	96.04
Average size of a firmware	274.7MB
Android Version ($\geq$4.0)	210
Phone Models (e.g., Samsung Galaxy S4)	101

Table 1: Summary Statistics of 250 Firmwares

4. SECURITY EVALUATION OF PRE-INSTALLED APPLICATIONS

In this section, we briefly introduce how to pre-install and extract applications in the Android firmware, then we show our analysis methodologies and experiment results for Android pre-installed applications.

4.1 Introduction to Pre-installation

There are two methods to pre-install the applications in the Android firmware: One is putting the apk files to the "`/system/app`" folder or "`/data/app`" folder of the firmware. Then the system will pre-install these applications when it launches. The advantage of using this method is that the applications in the "`/system/app`" folder (not the "`/data/app` folder) can gain the system privilege. The drawback of this approach is that the applications cannot execute normally if it contains any native libraries i.e., libraries which are written in C/C++, such as `foo.so`. Because the Android system does not extract the native libraries for applications in the "`/system/app`" folder, the applications cannot find the related native libraries when it executes.

The other way is using the "`pm install`" command to create a pre-install script and add the trigger script to the "`init.rd`" file in the "`boot.img`" file. After that, the system will trigger the pre-install shell script the first time it

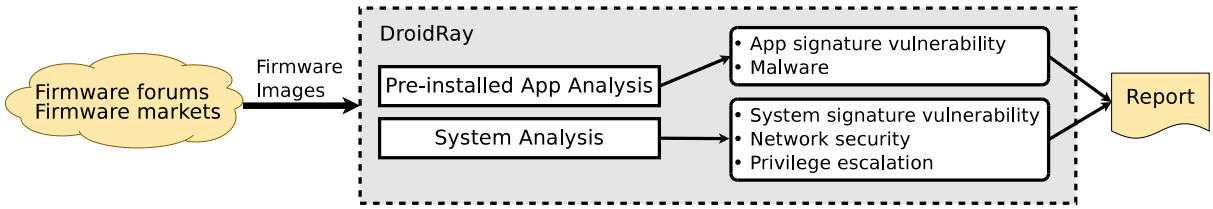

Figure 2: Architecture of DroidRay

launches. The advantage of using this method is it uses the normal installation process, so the application which contains native libraries can execute normally. The drawback is the pre-installed applications cannot obtain the system privilege.

In order to analyze the pre-installed applications, we need to extract all the `apk` files from the Android firmware. In DroidRay, our system uses both static and dynamic approaches to extract `apk` files. Since an `apk` file is in fact a zip file which includes `AndroidManifest.xml` and `classes.dex` file. The static approach of extracting `apk` files is to uncompress the Android firmware and extract all the files which has the zip magic number and contains `AndroidManifest.xml` file. The dynamic approach of extracting `apk` files is to flash Android firmware into the device and then use "`pm list packages`" and "`pm path`" command to obtain all the paths of these `apk` files in the device, and then use "`adb pull`" to extract all the `apk` files.

4.2 Application Signature Vulnerability Detection

In Android, application signature is very important for security and privacy. The Android system requires all installed applications be digitally signed with a certificate and the private key should be held by the application's developer. The Android system uses the certificate as a mean of identifying the author of an application and establishing trust relationships between applications. Note that if two applications have the same package name and same signature, they can replace each other due to the application update mechanism. In addition, if two applications are signed by the same certification and claim the same "`SharedUserId`" attribute in their `AndroidManifest.xml` files, they will share the same union permission set. Furthermore, with the same uid, they will have the permission to access each other's data. Therefore, if two applications have the same certificate but one of them is a malware, this implies the malware has penetrated into the system.

AOSP (Android Open source Project) comes with four signature key pairs in the "`build/target/product/security/`" folder. The platform key is used to sign core parts of the system. The shared key is used to sign home and contacts part of AOSP. The media key is used to sign media and download framework parts of AOSP. The test key is used to sign everything else. These signature key pairs are for the developers to build *test* Android firmwares. The signature information can be found in Table 2. The first column shows the MD5 value of the signature after signed. The second column shows the description of the signature. Because all of these four key pairs are created by the Android team, they have the same description. The last column shows the signature of file names. It is important for us to point out that developers should *not* use these default AOSP key pairs to build their released Android applications because everyone can obtain these key pairs from the AOSP. Therefore, it is in danger that an application with "`SharedUserId`" attribute is signed by the AOSP keys.

In order to detect application signature vulnerability, DroidRay first extracts the "`SharedUserId`" attribute from the `AndroidManifest.xml` file and the signature information from the `RSA` file of the application. Then DroidRay puts all of the information into the database and compares with the default signatures of AOSP. Note that we did not count the applications which share the same uid (as known as "`android.uid.system`") with the system platform. The reason is that these applications belong to the platform applications with the system level privilege, so we classify them as a part of system signature vulnerability. We will discuss this form of security analysis in Section 5.

Among 24,009 apps in 250 Android firmwares we analyzed, we discover that 5,712 apps have the "`SharedUserId`" attribute in their `AndroidManifest.xml` files. Moreover, 1,947 (34%) applications are signed by the default AOSP key pairs. Table 3 shows how many shared uid applications are signed by which key pairs (because of the page limit, the full list can be found at `https://www.dropbox.com/s/f7c5de4s55o2rk9/AppSign.zip`). This is quite alarming since malware writers can easily sign their malware with the default AOSP key pairs, and the malware can gain extra permissions and modify other applications' data if they have the same shared uid. For example, we discover that all the firmwares released by Cyanogenmod [22] have several vulnerable applications. In particular, `com.android.providers.media`, `com.android.providers.drm` and `com.android.providers.downloads` are signed by the default AOSP media key and they all have a same shared uid, "`android.media`". Also, `com.android.contacts`, `com.android.providers.contacts`, `com.android.providers.applications` and `com.android.providers.userdictionary` are signed by the default AOSP shared key and they all have a same shared uid, "`android.uid.shared`". Therefore, if the malware is signed by the default AOSP key (media or shared key) and claims the same shared uid ("`android.media`" or "`android.uid.shared`"), it can access all the data and get permissions (e.g., `READ_CONTACTS`, `WRITE_CONTACTS`, `WRITE_SETTINGS` and `ACCESS_DRM`) from vulnerable applications.

4.3 Malware Detection

DroidRay first uses anti-virus software (e.g. VirusTotal [11]) to scan all apks and filter out the "known" and common malware. For the remaining applications, DroidRay only retains the applications which have dangerous permissions (e.g., sending SMS message) or silent installation behavior. For the applications with dangerous permissions, DroidRay uses permission-to-api map table [1] to find the code section which uses the related API of dangerous per-

MD5 of Signature	Signature Description	Signature Type
8ddb342f2da5408402d7568af21e29f9	EMAIL=android@android.com,CN=Android,OU=Android, O=Android,L=Mountain View,ST=California,C=US	platform
e89b158e4bcf988ebd09eb83f5378e87	EMAIL=android@android.com,CN=Android,OU=Android, O=Android,L=Mountain View,ST=California,C=US	testkey
1900bbfba756edd3419022576f3814ff	EMAIL=android@android.com,CN=Android,OU=Android, O=Android,L=Mountain View,ST=California,C=US	media
5dc8201f7db1ba4b9c8fc44146c5bcc2	EMAIL=android@android.com,CN=Android,OU=Android, O=Android,L=Mountain View,ST=California,C=US	shared

Table 2: Default Signature Information

MD5 of Signature	Signature Description	# of App
cde9f6208d672b54b1dacc0b7029f5eb	CN=Android,OU=Android,O=Google Inc.,L=Mountain View,ST=California,C=US	767
8ddb342f2da5408402d7568af21e29f9	emailAddress=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	673
5dc8201f7db1ba4b9c8fc44146c5bcc2	emailAddress=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	518
1900bbfba756edd3419022576f3814ff	emailAddress=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	503
4a441695cb20427d284e7ded135925ad	emailAddress=android@htc.com,CN=Android,OU=Android,O=Android,L=Taoyuan,ST=Taoyuan,C=TW	268
f11fdf766b78025bf4035cb1b7ad483c	emailAddress=android.os@samsung.com,CN=Samsung Cert,OU=DMC,O=Samsung Corporation,ST=South Korea,C=KR	263
2eed75e85b154fb0c1013ecd16115c84	CN=Tencent,OU=Tencent,O=Tencent,L=Beijing,ST=Beijing,C=CN	255
e89b158e4bcf988ebd09eb83f5378e87	emailAddress=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	253
d087e72912fba064cafa78dc34aea839	emailAddress=android.os@samsung.com,CN=Samsung Cert,OU=DMC,O=Samsung Corporation,ST=South Korea,C=KR	241
ea75f0b73ee288cd683b7a11716b9f77	emailAddress=android@htc.com,CN=Android,OU=Android,O=Android,L=Taoyuan,ST=Taoyuan,C=TW	159
f31d93c0e9064bafa39cbe653360e6ba	emailAddress=android.os@samsung.com,CN=Samsung Cert,OU=DMC,O=Samsung Corporation,ST=South Korea,C=KR	128
701478a1e3b4b7e3978ea69469410f13	emailAddress=miui@xiaomi.com,CN=MIUI,OU=MIUI,O=Xiaomi,L=Beijing,ST=Beijing,C=CN	115

Table 3: App Signature Information

missions and then check whether it has malicious behavior or not. On the other hand, in order to find those applications with silent installation behavior, DroidRay disassembles each application and uses two silent installation code patterns to search the assemble code. One code pattern we used is using the hidden installation API, `android.content.pm.PackageManager.installPackage()`, to install the `apk` file. The other code pattern we used is `Runtime.exec()` which is to execute "`pm install`" command to install the `apk` file. After determining the code section, we continue to check whether it contains malicious behavior or not.

Among the 250 Android firmwares we analyzed, we discovered that 19 firmwares contain malware. This means about 7.6% Android firmwares contain pre-installed malware. Again, this is quite alarming since many low-cost manufactures use public firmwares for their products. We provide more detailed information of these 19 firmwares in Table 4.

- We discover that 19 firmwares contain the same Android malware which is known as "Agent". This is a premium SMS trojan. In addition, it has a special permission, `android.permission.INSTALL_PACKAGES`, as compared with other Android malware. Usually, a normal Android application cannot be installed in the smartphone if it has the `android.permission.INSTALL_PACKAGES` permission, because this is a privilege permission only reserved for system applications. However, pre-installed applications are different. Because they are stored in the "`/system/app/`" folder, these apps belong to the system applications. Therefore they possess the `android.permission.INSTALL_PACKAGES` permission. Once the malware has this permission, it can install other Android applications into the smartphone without notifying the user that a new application is being installed.
- We discover "JSmsHider", which is a premium SMS trojan that uses the HTTP protocol to receive and execute the command from some botnet masters.
- We discover "Stesec", which is a SMS trojan which makes phone calls to long-distance numbers or sends SMS premium rate phone numbers without the phone owner's knowledge.
- We discover "Hippo", which is another premium SMS trojan that can cause additional phone charges by sending SMS messages to some hard-coded premium-rated numbers in the mobile application.
- One malware family we want to emphasize is the "CEPlugnew". In our security analysis, we discover that it uses various encryption techniques to bypass the antivirus detection and hide in the firmwares. To gain a deeper understanding on this malware family and its business chain, we carried out a detailed study of this new malware family. Results and discussion will be presented in Section 6.

5. SECURITY EVALUATION OF THE SYSTEM

In this section, we discuss how we perform system security detection. The process can be divided into three aspects: (1) *system signature vulnerability detection*, (2) *network security vulnerability detection*, and (3) *privilege escalation vulnerability detection*.

MD5 of Firmware Image	Size (MB)	OS Version	Phone Model	# of Apps	Malware Path	Malware Name
3643C59163D102616BF547A465837C44	934	4.1.2	GT-I9300	244	/optional/other/digua/preload/digua.apk	Agent
5B7E06BCE9920E82CB0A8CBC0302BC7E	215	4.0.3	ZTE U970	83	/system/framework/framework.jar	Agent
5BFE256A10BDB9A44BA7F7AF78DDD3AC	225	4.0.4	GT-I9300	83	/system/app/BaiduReader.apk/assets/huafubao.apk	Agent
6B20E70C4AD84B1EE5F840E920E718EB	183	4.0.4	HUAWEI C8812	82	/system/app/168_(2216G7).apk	Agent
76099C37D57AC1A5659B7B312155733C	185	4.2.2	Galaxy Nexus	78	/system/preload/digua.apk	Agent
8BA7722157471E31946D500BDAAA7998	1026	4.2.1	GT-I9300	249	/optional/other/digua/preload/digua.apk	Agent
959A7ADC6F1A8AC4D8D266C4AB3EDD9E	262	4.1.2	X907	85	/system/app/YXPhoneBook.apk	JSmsHider
979DAF30377A1D85CD19F1CE71A7A5DC	87	2.3.7	LT15i	18	/system/app/Youni_1.1.16_1103.apk	Hippo
99DD205A5E6D2045CE3D57D157278FBF	207	4.0.4	Incredible S	74	/system/etc/product/applications/duoku.apk	Agent
A364CF949574DAEF8C30072F45504432	83	2.3.7	Blade	55	/system/app/BaiduReader.apk	Agent
BC3B5944A758A8B7734C152E096C80A6	226	4.0.4	GT-I9300	86	/system/app/iReader.apk	Agent
BE541971F46F5BCE0EDC0A397B608FF9	387	4.1.2	LT26i	164	/system/app/BaiduReader.apk	Agent
BEC16CFCB0C5FBE9E6F8D402D88BDB8B	937	2.2.2	ZTE-U V880	74	system/app/SecuritySms.apk	Stesec
D3ECD1333E7F46CE3E028CEA1CB3F224	628	4.1.2	GT-I9300	170	/system/etc/product/applications/duoku.apk	Agent
E9F033CD8C694A5BD56C7AE94A425C97	406	4.2.1	ZTE N986	156	/system/app/BaiduReader.apk	Agent
F4DF6D64A4F856CA1B53B400A33C55F4	113	2.3.7	GT-I9000	67	/system/app/youni_2.2.2.3.apk	Hippo
F62C11C7827BBA0374DB69B276910266	161	4.1.2	GT-I9300	64	/system/app/Plugnew.apk	CEPlugnew
F893F4BE1A0A280BD62887A2BF2C36C6	193	4.2.2	SGH-I897	83	/system/preload/digua.apk	Agent
FD2EA6A845A86AAB0122BB2ADF90F2C1	81	2.3.7	Wildfire S	55	/system/app/iReader.apk	Agent

Table 4: Malicious firmwares with pre-installed apps

5.1 System Signature Vulnerability Detection

As mentioned in Section 4, developers should not use the AOSP default key pairs to build their released Android applications because anyone can get these key pairs from the AOSP. For Android firmwares, it also have two signature mechanisms for integrity verification and system level permission control. However, our experiment result shows that there exist many Android firmwares which were signed by the default AOSP key pairs and this will create two types of default signature security problem:

(1) Android firmware default signature problem: The Android firmware signature mechanism is similar to other Android applications. The signing program first calculates the SHA1 value of each file in the firmware, then it uses a private key to encrypt these SHA1 values. The signing program will put the encrypted SHA1 values and the public key in the "`META-INF`" folder of the firmware. Before the firmware can be flashed to an Android device, the recovery partition will use the encrypted SHA1 values and its public key to check the integrity of each file as well as to verify the signature of the firmware. Note that the recovery partition of some manufacturers' products (e.g., Samsung) will check whether the public key is indeed its own key or not. If it is not, one cannot flash the firmware to the flash memory. However, many users may choose to turn off the signature verification process so to install the third-party firmwares.

By analyzing all of our 250 downloaded firmwares, we discovered that 206 (82.4%) firmwares have the *same public key*, which is the test key of the default key pairs provided by AOSP. The result of our analysis is showed in Table 5. For example, the fifth row of the table shows the public key information of the AOSP and there are *206* firmwares using this key pairs. Note that if malware writers use the AOSP default key pairs to create firmwares, this will give people the false impression that the Android firmware was created by the official Android open source project.

(2) Android system default signature problem: The system signature information is stored at: "`/system/framework/framework-res.apk/META-INF/CERT.RSA`". By using the pkcs7 function of the "`openssl`" library, the system can parse the signature information from "`CERT.RSA`". Note that the Android system signature is different from the Android firmware signature. The Android firmware signature is used to check the integrity of the files in the firmware. But the system signature is used for permission control. If one application has the same public key as the system, it can use the system level permissions.

By analyzing all of our 250 downloaded firmwares, we discovered that 142 (56.8%) Android firmwares have the *default public keys*, which are part of the default key pairs provided by AOSP [10]. This is alarming because if a malware writer uses the AOSP key pairs to sign their malware, then the malware can get the system level privilege. With the system level privilege, the malware can silently install new applications and modify other applications. The result of our analysis is showed in Table 6 (because of the page limit, the full list can be found at `https://www.dropbox.com/s/8fbl0yujqoaja1h/SystemSign.xls`). For example, the first row of the table shows the public key information of the AOSP and there are 134 Android firmwares using this key pairs.

5.2 Network Security Vulnerability Detection

Android is a derivative based on a modified Linux 2.6 with a Java programming interface, so it has components (e.g., hosts and iptables) similar to those of Linux to control network filtering and forward. In this subsection, we present the security issues of these components.

(1) Hosts security: The `hosts` file is a plain text file used by an operating system to map host names to IP addresses. For example, if the user wants to visit "`www.google.com`" and if there is a map record in the "`/etc/hosts`" file, the system will not request IP lookup from a DNS server but instead, it directly uses the IP address which is recorded in the `hosts` file. Therefore, it would be very dangerous if the hackers modify this file and add malicious redirection information. For example, if a user wants to visit some legitimate websites listed in the "`/etc/hosts`" file, a compromised system will

MD5 of Signature	Signature Information	# of Firmware
1f1b2fd4e0bded6ca5088afa287b5332	EMAIL=mobile@zte.com.cn, CN=zte.com.cn, OU=Product R&D Center I, O=ZTE Corporation, L=Nanjing, C=CN	1
5fd1d7579992cff701ea20873e8ac99f	CN=Smaritsan	2
ab82cce8af68057b7ab8867b5dd75d65	EMAIL=Android@amoi.com.cn, CN=Amoi, OU=Amoi, O=Amoi, L=Xiamen, ST=Fujian, C=CN	1
943e1151975cfd342a95f2a7512bbff8	EMAIL=12001@oppo.com, CN=12001, OU=M6, O=OPPO, L=DongGuang, ST=GuangDong, C=CN	1
e89b158e4bcf988ebd09eb83f5378e87	EMAIL=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	206
fcf645259f609183606ab55da67fc9db	EMAIL=oppo@oppo.com, CN=SmartPhone, OU=PLF, O=OPPO, L=ChangAn, ST=DongGuan, C=CN	2
fe3c74f0431ea0dc303a10b6af6470fc	EMAIL=bms@baidu.com,CN=BMS,OU=BaiduYi,O=Baidu,L=Haidian,ST=Beijing,C=CN	9
No Signature	No Signature Information	28
Total		250

Table 5: Firmware signature count with highlighted row indicating misleading signature

MD5 of Signature	Signature Information	# of Firmware
8ddb342f2da5408402d7568af21e29f9	emailAddress=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	134
d087e72912fba064cafa78dc34aea839	emailAddress=android.os@samsung.com,CN=Samsung Cert,OU=DMC,O=Samsung Corporation,ST=South Korea,C=KR	24
2eed75e85b154fb0c1013ecd16115c84	CN=Tencent,OU=Tencent,O=Tencent,L=Beijing,ST=Beijing,C=CN	13
701478a1e3b4b7e3978ea69469410f13	emailAddress=miui@xiaomi.com,CN=MIUI,OU=MIUI,O=Xiaomi,L=Beijing,ST=Beijing, C=CN	12
4abc4868a502c9c77b50ca9deb53b672	emailAddress=yi@baidu.com,CN=rom-platform,OU=BaidYi,O=Baidu,L=Haidian,ST=Beijing, C=CN	8
bb7bce1b1090fc3a6b67ebc88701acdd	emailAddress=mobile@huawei.com,CN=AndroidTeam,OU=TerminalCompany,O=Huawei, L=Shengzhen, ST=Guangdong,C=CN	9
0f92c3b692319a1daabac6c94b07fbee	emailAddress=Admin@dianxinos.com,CN=TAPAS,OU=SDG,O=TAPAS,L=BeiJing,ST=BeiJing, C=CN	5
10637522aa8f840170c50fcbe61227ac	CN=Smartisan	4
9e5b0111e0408bb405c819ea0784b69d	CN=Sony_Ericsson_E_Platform_Signing_Live_864f,O=Sony Ericsson Mobile Communications AB, C=SE	7
a7babf3e3872080766e5331943aafa58	emailAddress=android@htc.com,CN=Android,OU=Android,O=Android,L=Taoyuan,ST=Taoyuan, C=TW	6
75711ca27d4693cc27432c47ac8582c0	emailAddress=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	5
d4c6ea170fd9ef95572ab50318b695de	emailAddress=oppo@oppo.com,CN=SmartPhone,OU=PLF,O=OPPO,L=ChangAn,ST=DongGuan,C=CN	4
2bf07615199371b0c35ef2e23936b345	CN=lidroid,OU=lidroid,O=lidroid,L=xi'an,ST=shaanxi,C=CN	3
e89b158e4bcf988ebd09eb83f5378e87	emailAddress=android@android.com,CN=Android,OU=Android,O=Android,L=Mountain View,ST=California,C=US	3
8701cf44ad2623c4a5c89944b24d6417	CN=lidroid.com,OU=lidroid.com,O=xian,L=shaanxi,ST=SX,C=86	2
...	...	...
Total		250

Table 6: System signature count with highlighted row indicating AOSP default signatures

not connect the user to the legitimate server but instead, requests will be re-routed to some malicious websites.

To investigate this form of network security vulnerability, we use DroidRay to extract the `hosts` file from Android firmware and check whether it has malicious modifications or not. Note that some developers may use `hosts` file to block advertisements. They redirect the advertisement servers' host names to "127.0.0.1" in order to prevent the server from sending advertisement to the smartphone. In this case, we will treat it as a warning but not a malicious behavior. By analyzing all of our 250 downloaded firmwares, we discovered that 54 firmwares modified their `hosts` files and five Android firmwares contained malicious `hosts` file (please refer to Table 7). For example, the first row of Table 7 shows that "*HUAWEI*" Android firmware with 4.1.2 version has malicious `hosts` file which redirects Google and Youtube websites to malicious websites.

(2) Iptables security vulnerability: `iptables` is used to set up, maintain, and inspect the tables of IPv4 or IPv6 packet filter rules. It is more versatile than the `hosts` file specification because it can define many forwarding/filtering rules for each packet. Therefore, it is very dangerous if hackers add malicious rules in the `iptables`. In order to detect this form of vulnerability, we use DroidRay to perform the analysis. For the static analysis, DroidRay first searches all the initial scripts in the firmware and checks whether it has "`iptables`" command or not. Then DroidRay searches `iptables` configuration file from the "/system/etc/" folder. For the dynamic analysis, DroidRay executes the "`iptables -list-rules`" command to obtain all rules from the devices. If the system finds that the `iptables` rules are not none, DroidRay will report a warning to the analysts. Note that DroidRay will not consider the forward rules that redirect the advertisement servers' package to "127.0.0.1". By analyzing all of our 250 downloaded firmwares, we discovered that *two* Android firmwares modified their `iptables` rules and *no* Android firmware contains malicious `iptables` rules.

5.3 Privilege Escalation Vulnerability Detection

Privilege escalation vulnerabilities can be exploited by a malicious application to gain high level privileges (e.g., root and system levels) and execute malicious actions that would normally be restricted by the Android system. A number of such vulnerabilities (e.g., GingerBreak [2] and MasterKey [26]) were discovered in the core Android platform, affecting nearly all Android devices. Although manufacturers have tried their best to fix these vulnerabilities, unfortunately, privilege escalation vulnerabilities still remain unpatched on large populations of Android firmwares.

MD5 of firmware Image	Size (MB)	OS Version	Phone Model	# of Apps	Malicious Behavior
5EC3A725107D787C51F73055E2B2836B	473	4.1.2	HUAWEI U9508	127	Redirect google and youtube
5F262C7046904152ABC0DC0635694EA2	286	4.0.3	X907	92	Redirect facebook
67C031B8BCC97B2EBE672B30243C3050	306	4.0.3	X907	92	Redirect yahoo
8A3598500340843FF1D0609F42E852E7	428	4.1.2	LT26i	161	Redirect google
E43B91728003193EA4E889C20310FD5D	207	4.0.4	HUAWEI C8812	75	Redirect adobe and wikipedia

Table 7: Modified Firmwares with Malicious Hosts File

We use DroidRay to explore the existence of privilege escalation vulnerabilities on Android firmware systems. For the static analysis, we classify escalation vulnerabilities into *native level vulnerabilities* (e.g.,GingerBreak and ZergRush [24]) and *Java level vulnerabilities* (e.g., MasterKey). For native level vulnerabilities, DroidRay first extracts the potential vulnerable `elf` files from the Android firmware and then disassembles the potential vulnerable functions into ARM assemble language. Then the system compares the control flow of the potential vulnerable functions with the vulnerable functions stored in our vulnerability database. Note that for patching vulnerable files, developers need to add new judgement code into the vulnerable functions. This is the reason why we choose the control flow as the signature. For example, Android 2.2 (as known as Froyo) has a *RageAgainsttheCage* vulnerability. This vulnerability exploits the lack of checking for the return value of `setuid()` function in the `/system/bin/adb` file. After the initial process has the root privilege, the `adb` daemon attempts to call `setuid()` to set its uid to the shell privilege in its code:

```
setgid(AIL_SHELL);
setuid(AIL_SHELL);
```

However, the exploit attempts to fork as many "*adb*" processes as possible in order to make the "*setuid*" fail. In addition, the current `adb` code does not check whether the `setuid()` call was successful or not and it will keep running as a root process even if the call fails. To patch this vulnerability, Android 2.3 adds the privilege check after the `setuid()` call:

```
if (setgid(AIL_SHELL) != 0) {
    exit(0);
}
if (setuid(AIL_SHELL) != 0) {
    exit(0);
}
```

For Java level vulnerabilities, DroidRay first extracts the potential vulnerable `odex` or `jar` files from the Android firmware and then disassembles the potential vulnerable functions into Dalvik assemble language. Then the system compares the control flow of the potential vulnerable functions with the vulnerable functions stored in our vulnerability database. For example, Android 4.2 has a master key vulnerability. This vulnerability exploits the lack of checking for the duplicate name of the entries in the zip file. So hackers can create two files with the same name in order to bypass the signature verification:

```
LinkedHashMap<String, ZipEntry> mEntries
    = new LinkedHashMap<String, ZipEntry>();
for (int i = 0; i < numEntries; ++i) {
    ZipEntry newEntry = new ZipEntry(hdrBuf, bin);
    mEntries.put(newEntry.getName(), newEntry);
}
```

To patch this vulnerability, Android 4.3 adds the duplicate name check after the `mEtries` putting the name into the HashMap:

```
for (int i = 0; i < numEntries; ++i) {
    ZipEntry newEntry = new ZipEntry(hdrBuf, bin);
    String entryName = newEntry.getName();
    if (mEntries.put(entryName, newEntry) != null) {
        throw new ZipException("Duplicate" +
        "entry␣name:␣" + entryName);
    }
}
```

For dynamic analysis, DroidRay uses the "`adb`" to execute all real exploits in our vulnerability database and then checks whether the exploits are successful or not. Last but not least, DroidRay reports the potential privilege escalation vulnerabilities to the user.

Currently, our vulnerability database includes four native level vulnerabilities and three Java level vulnerabilities (please refer to Table 8). In Table 8, the first column shows the name of the vulnerability. The second column shows the check points of the vulnerability and the last column shows the type (native or Java) of the vulnerability. For example, the first row shows the *RageAgainsttheCage* vulnerability we discussed before. By analyzing all of our 250 downloaded firmwares, we discovered that most of firmwares can defend against native level vulnerabilities. However, nearly 99% firmwares have Java level vulnerabilities. The experiment result is summarized in Table 9. The first column is the name of the vulnerability. The second column indicates how many firmwares have this type of vulnerability, and the last column depicts the percentage of vulnerable firmwares.

vulnerability name	# of firmware	Percentage
RageAgainsttheCage [3]	1	0.4%
GingerBreak [2]	2	0.8%
ZergRush [24]	40	16.0%
CVE-2009-1185 [6]	40	16.0%
Masterkey1[26]	249	99.6%
Masterkey2[25]	249	99.6%
Masterkey3[27]	250	100.0%

Table 9: Experiment result of privilege escalation vulnerabilities

6. PRE-INSTALLED MALWARE CASE STUDY

As mentioned in Section 4, we discovered a malware named "CEPlugnew" in numbers of firmwares. In March of 2013, we captured this pre-installed zero-day Android malware family

vulnerability name	Check Point	Type
RageAgainsttheCage [3]	adb_main() function of /system/bin/adb elf file	native
GingerBreak [2]	handlePartitionAdded() function of /system/bin/vold elf file	native
ZergRush [24]	dispatchCommand() function of libsysutils.so elf file	native
CVE-2009-1185 [6]	uevent_kernel_multicast_recv() function of /system/bin/sysinit elf file	native
Masterkey1[26]	readCentralDir() function of java.util.zip.ZipFile class of /system/framework/core.jar	Java
Masterkey2[25]	getInputStream() function of java.util.zip.ZipFile class of /system/framework/core.jar and ZipEntry() function of java.util.zip.ZipEntry class of /system/framework/core.jar	Java
Masterkey3[27]	getInputStream() function of java.util.zip.ZipFile class of /system/framework/core.jar	Java

Table 8: vulnerability Description

and its management system. The system is used to send premium rate SMS and distribute pay per install applications. In this section, we present our investigation of this zero-day Android malware, its operating process and its business chain.

6.1 Zero-day Malware: CEPlugnew

As we mentioned in Section 4, malware writers can pay money to some smartphone manufacturers in order to pre-install apps in smartphones. By using DroidRay, we studied several firmwares of low-cost smartphones and successfully discovered a pre-installed zero-day malware. We call this malware `CEPlugnew` because its package name is `com.example.plugnew`. Firstly, it has `INSTALL_PACKAGES` and `SEND_SMS` permission, so DroidRay extracts it from firmware for further analysis. Secondly, by using dynamic analysis, we find it is an advanced malware using the `DexClassLoader` to dynamically decrypt and load an encrypted JAR file. In addition, because it is a pre-installed app with system privilege, it can download and install any application when it receives commands from the malware server. By reverse engineering, we were able to determine the malware servers' addresses (e.g., 42.121.120.*). Furthermore, we discover that one of their servers has the MSSQL SQL injection vulnerability. We exploit this vulnerability and obtain a large database and its malware manage system. Let us present our findings.

6.2 Malware Management System

The malware management system we obtained is a web-based system to manage and view the status of premium rate SMS messages and silent installation of mobile applications. The system was written in JSP and was created in 2009. At first, this system was used to control the Symbian-based malware. According to the modification record, the system was modified and began controlling Android-based malware in 2011. Firstly, this system has a dynamic web page, the malware on the controlled smartphones will visit this dynamic web page on the specified time with some essential parameters (e.g., IP address, time, IMEI, IMSI, version, and phone model). Then the system will record the information in the MSSQL database. Secondly, if the malware writers want to distribute an application, they only need to create a new task and enter parameters for this task (e.g., the download link and distribution numbers). On the other hand, the malware on the controlled smartphone will request the server for the task, and execute it on the specified time. Thirdly, this system can be used to send a large number of SMS messages to a host of phone numbers. In addition, malware writers can use this system to check their monthly profits from different telecommunications operators.

6.3 Database Analysis

We obtain the database snapshot from the malware writers' server. The database includes more than 348,018 unique International Mobile Equipment Identification (IMEI) and 415,607 unique International Mobile Subscriber Identification (IMSI). The statistics of the database is summarized in Table 10. We find that all the smartphones in the database were infected within five months (from 2012.10 to 2013.03). In addition, by querying the database on the IP distribution map, we can approximate the locations of these IP addresses. Table 11 shows the number of infected IP addresses in countries. Note that due to the page limit, we only list those countries which have more than 50 infected smartphones. Finally, we use Google map API [14] to generate an infected smartphone distribution map and it is depicted in Figure 3. From the map, we can see that some major cities in China, like Guangzhou, Beijing and Chengdu, have the high number of victims. Other countries like US and other European countries are not immune either since they have infected devices. To the best of our knowledge, this is the first real world case about the pre-installed malware which has more than 348,018 infected devices.

Database Information	Description
Unique IMEI	348,018
Unique IMSI	415,607
# of Phone Model	3,178
Unique IP Address	568,072
First log time	2012-10
Last log time	2013-03

Table 10: Statistics of infected smartphones

7. RELATED WORK

In this section, we present some related works which are related to Android malware, Android application security and Android firmware security.

In [8], authors presented the `ded` decompiler and carried out a study of 1,100 Android applications. Their analysis framework allows researchers to observe not only the existence of dangerous functionality, but also how it occurs within an Android app. Felt et al. studied 18 Android malware in [9]. Jia et al. [18] conducted a study on the run-time enforcement of information-flow. In [33], authors discussed how to enhance users' comprehension of Android permissions. ADAM [35] and DroidChameleon [23] are two systems to generate Android malware variants. [40] was the first paper to provide a systematic study on the detection of malicious applications on Android Markets. The authors successfully discovered 211 malware and carefully analyzed the malicious behaviors of the malware. DroidMOSS [37] is an Android system to detect repackaged applications using

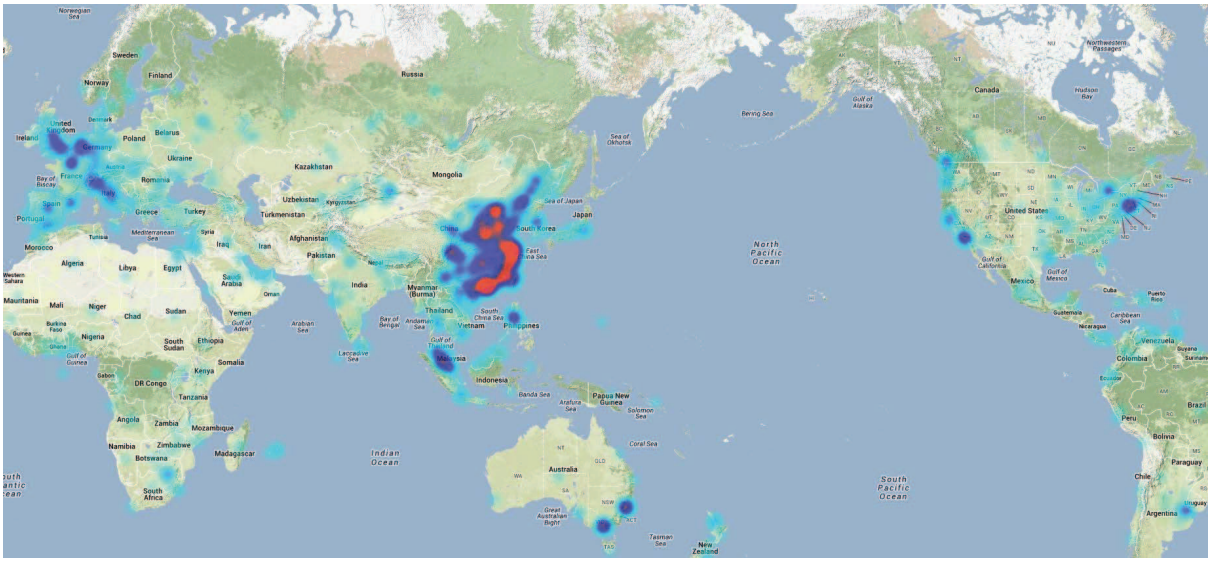

Figure 3: Distribution of infected Android devices around the world due to the pre-installed malware "CEPlugnew"

Country	# of Infected IP Addresses
CN-(CHINA)	561,657
TW-(TAIWAN)	1,299
HK-(HONG KONG)	891
KR-(REPUBLIC OF KOREA)	453
US-(UNITED STATES)	385
MY-(MALAYSIA)	305
VN-(VIET NAM)	202
RU-(RUSSIAN FEDERATION)	168
VE-(VENEZUELA)	165
MO-(MACAU)	159
SG-(SINGAPORE)	155
IT-(ITALY)	141
AU-(AUSTRALIA)	126
CA-(CANADA)	114
ES-(SPAIN)	105
GB-(UNITED KINGDOM)	105
TH-(THAILAND)	88
FR-(FRANCE)	81
DE-(GERMANY)	74
PH-(PHILIPPINES)	66
BG-(BULGARIA)	62
KZ-(KAZAKHSTAN)	62
ID-(INDONESIA)	61
NZ-(NEW ZEALAND)	58
AR-(ARGENTINA)	57
SA-(SAUDI ARABIA)	50

Table 11: Number of infected IP addresses in countries

fuzzy hashing. ContentScope [39] aimed to find the passive content leaks and pollution in Android applications. SmartDroid [34] can automatically reveal the UI-based trigger conditions. In [38], Zhou et al. studied characterization and evolution of Android malware with 1,260 samples. Kevin McNamee [21] showed us how to build a spyphone. However, all of the above mentioned works only focused on the detection and functionalities of Android malware and they did not focus on Android firmware security.

Woodpecker [12] analyzed each application in a smartphone firmware to explore the reachability of a dangerous permission from a public, unguarded interface. Lei et al. [20] studied ten representative stock Android images from five popular smartphone vendors and they found a lot of security problems on the permission usages of pre-installed applications. However, all of the above mentioned works only focused on the applications. XRay[28] is an app for scanning security vulnerabilities of Android system, so it cannot scan the firmwares. Our paper is the first detailed study on both pre-installed application security and system security for Android firmwares. In particular, we have done large scale experiments on 250 Android firmwares and performed a detailed case study on infected firmwares.

8. CONCLUSION

As malware writers are using firmwares to spread new malware, there is an urgent need to combat this new form of distribution channel. We present the design and implementation of "*DroidRay*": a security evaluation system for customized Android firmwares. The system uses both static and dynamic analyses to evaluate the firmware security on application level and system level. We carry out a comprehensive study on 24,009 pre-installed applications and 250 Android firmware systems, and discover compromised firmwares can contaminate the system and inject new malware into devices. We found 1,947 (8.1%) pre-installed applications have signature vulnerability and 19 (7.6%) firmwares contain pre-installed malware. In addition, we discovered 142 (56.8%) Android firmwares have the default signature vulnerability, five (2.0%) Android firmwares contain malicious *hosts* file, at most 40 (16.0%) Android firmwares have native level privilege escalation vulnerability and at least 249 (99.6%) Android firmwares have Java level privilege escalation vulnerability. In particular, we carry out detailed investigation on a real world pre-installed zero-day Android malware known as `CEPlugnew`, which involves 348,018 infected Android smartphones and reveal its geographical spread.

9. REFERENCES

[1] F. Adrienne, Porter, C. Erika, H. Steve, S. Dawn, and W. David. Android permissions demystified. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2011.

[2] C-skill. Gingerbreak. `http://c-skills.blogspot.hk/2011/04/yummy-yummy-gingerbreak.html`, 2011.

[3] C-skill. Rageagainstthecage. `https://github.com/bibanon/android-development-codex/wiki/rageagainstthecage`, 2011.

[4] A. Central. Android central (rom market). `http://www.androidcentral.com/tags/firmware`, 2013.

[5] C. Daily. Low-end smartphone fight. `http://www.chinadaily.com.cn/bizchina/2012smartphone/2012-07/16/content_15703750.htm`, 2012.

[6] N. V. Database. Cve-2009-1185. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1185`, 2009.

[7] DownloadAndroidROM. Downloadandroidrom (rom market). `http://downloadandroidrom.com/`, 2013.

[8] 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, 2011.

[9] 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*, 2011.

[10] Google. Android open source project. `http://source.android.com`, 2008.

[11] Google. virustotal. `https://www.virustotal.com/`, 2013.

[12] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012.

[13] I. D. C. (IDC). Apple cedes market share in smartphone operating system market as android surges and windows phone gains, according to idc. `http://www.idc.com/getdoc.jsp?containerId=prUS24257413`, 2013.

[14] G. Inc. Google map apis. `http://developers.google.com/maps/`, 2012.

[15] L. Inc. Security alert: Malware found targeting custom roms (jsmshider). `https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting-custom-roms-jsmshider/`, 2011.

[16] Ipmart. Ipmart (rom forum). `http://www.ipmart-forum.com/forum.php`, 2013.

[17] J. Janego. The security implications of custom android roms. `http://labs.neohapsis.com/2011/12/21/the-security-implications-of-custom-android-roms/`, 2012.

[18] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on Android (extended abstract). In *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*, Sept. 2013.

[19] Jidi. Rom jidi (rom market). `http://www.romjd.com/`, 2013.

[20] W. Lei, G. Michael, Z. Yajin, W. Chiachih, and J. Xuxian. The impact of vendor customizations on android security. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2013.

[21] K. McNamee. How to build a spyphone. `Blackhat`, 2013.

[22] C. Mod. Cyanogen mod (rom forum). `http://www.cyanogenmod.org/`, 2013.

[23] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the ASIACCS '13*.

[24] Revolutionary. Zergrush. `http://forum.xda-developers.com/showthread.php?t=1296916`, 2011.

[25] saurik. Android bug superior to master key. `http://www.saurik.com/id/18`, 2009.

[26] saurik. Exploit and fix android master key. `http://www.saurik.com/id/17`, 2009.

[27] saurik. Yet another android master key bug. `http://www.saurik.com/id/19`, 2009.

[28] D. Security. Xray for android. `http://www.xray.io/#vulnerabilities`.

[29] Shendu. Rom shendu (rom market). `http://www.shendu.com/android/`, 2013.

[30] VR-ZONE. Research shows chinese manufacturers account for 20 percent of smartphones worldwide, india on the rise. `http://vr-zone.com/articles/research-shows-chinese-manufacturers-account-for-20-percent-of-smartphones-worldwide-india-on-the-rise/49868.html`, 2013.

[31] D. Walker. Pay-per-install pays big bucks in the mobile world. `http://www.scmagazine.com/pay-per-install-pays-big-bucks-in-the-mobile-world/article/258731/`, 2012.

[32] Xda. Xda-developers (rom forum). `http://www.xda-developers.com/`, 2013.

[33] L. Yang, N. Boushehrinejadmoradi, P. Roy, V. Ganapathy, and L. Iftode. Short paper: enhancing users' comprehension of android permissions. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '12, 2012.

[34] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '12, 2012.

[35] 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, 2013.

[36] Zhijia. Rom zhijia (rom market). http://www.romzj.com/, 2013.

[37] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012.

[38] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[39] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.

[40] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012.

[41] H. Z. Zihang Xiao, Qing Dong and X. Jiang. Oldboot: the first bootkit on Android. http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/, 2014.