Towards Discovering and Understanding Unexpected Hazards in Tailoring Antivirus Software for Android

Heqing Huang[†], Kai Chen[‡], Chuangang Ren[†], Peng Liu[†], Sencun Zhu[†], Dinghao Wu[†]
[†]The Pennsylvania State University, University Park, PA, USA
[‡]Institute of Information Engineering, Chinese Academy of Sciences, China
{hhuang, szhu, cyr5126}@cse.psu.edu, {chenkai}@iie.ac.cn, {pliu, dwu}@ist.psu.edu

ABSTRACT

In its latest comparison of Android Virus Detectors (AVDs), the independent lab AV-TEST reports that they have around 95% malware detection rate. This only indicates that current AVDs on Android have good malware signature databases. When the AVDs are deployed on the fast-evolving mobile system, their effectiveness should also be measured on their runtime behavior. Therefore, we perform a comprehensive analysis on the design of top 30 AVDs tailored for Android. Our new understanding of the AVDs' design leads us to discover the hazards in adopting AVD solutions for Android, including hazards in malware scan (*malScan*) mechanisms and the engine update (*engineUpdate*). First, the *malScan* mechanisms of all the analyzed AVDs lack *comprehensive* and *continuous* scan coverage. To measure the seriousness of the identified hazards, we implement targeted evasions at certain time (e.g., end of the scan) and locations (certain folders) and find that the evasions can work even under the assumption that the AVDs are equipped with "complete" virus definition files. Second, we discover that, during the *engineUpdate*, the Android system surprisingly nullifies all types of protections of the AVDs and renders the system for a period of high risk. We confirmed the presence of this vulnerable program logic in all versions of Google Android source code and other vendor customized system images.

Since AVDs have about 650–1070 million downloads on the Google store, we immediately reported these hazards to AVD vendors across 16 countries. Google also confirmed our discovered hazard in the *engineUpdate* procedure, so feature enhancements might be included in later versions. Our research sheds the light on the importance of taking the secure and preventive design strategies for AVD or other mission critical apps for fast-evolving mobile-systems.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection–*Invasive software; Access controls*

Keywords

Mobile; Malware; Anti-malware; Vulnerability measurement

ASIA CCS'15, April 14–17, 2015, Singapore..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714589 .

1. INTRODUCTION

The increasing popularity of mobile computing devices (e.g., smartphones and tablets) attracts both normal users and malware writers. Among the popular mobile platforms, Android has not only conquered a lion's share of the market, but also gained 98.1% share of mobile malware detected in 2013 [13]. Many companies, including those reputable ones focusing on PC security as well as some new startups, have turned their attention to mobile platform security and tailored their antivirus detectors particularly for Android [2]. Every three months, the independent antivirus test lab, AV-TEST, generates a report [6], comparing the detection rates of Android Virus Detectors (AVDs). The latest report indicated that the malware signature databases (MSD) of popular AVDs under test have achieved on average about 95% high malware detection rate. However, since Android allows executing both ARM binaries and Dalvik bytecode file (e.g., *.so* and *.dex* files) from dynamically loaded payload files to better serve application (app) developers at runtime [43], it has been shown that more malware [53, 52] based on the dynamic code loading are created and spread through the widely used Android app repackaging techniques [26, 34, 32, 48, 49]. Therefore, besides evaluation based on the quality of the MSDs, the success of the AVD's design on the fast-evolving Android platform must also be measured based on its efficacy in identifying malware's dynamic behavior (e.g., dynamic malicious payloads dropping/executing).

In this work, we conduct the first comprehensive analysis of top 30 AVDs (listed in Table 1), which currently has a total of 650-1070 million user downloads for the Google-Play store alone. Previous works [44, 40] have only focused on evaluating the quality of Virus Definition File (VDF) for virus detectors based on the well-known AVD weaknesses (e.g., vulnerable to transformation attacks). We take it a step further and our analysis concentrates on the scanning mechanism of Android AVDs.

Protection Assumption I: We assume Android AVD vendors have developed a *complete* MSD, which has been updated with all the reported malicious payloads and all the possible obfuscated versions of the payloads. Note that under this assumption, the Android AVDs are immune to common transformation attacks [44, 40]. Bearing this assumption in mind, we would like to understand if there are still deficiencies in the malware scan (*malScan*) mechanism itself that can cause potential hazards.

Hazards in malScan. The *malScan* operation here means the general malware recognition mechanism, which fingerprints the malware based on the VDF updated from the MSD. Generally, there are two types of *malScan* operations for Android: *light monitoring malScan* and *heavy sweeping malScan*. Four out of the top 30 AVDs have implemented the *light monitoring malScan*, which leverages the APIs from the Android *FileObserver* class. This en-

Table 1: Popular Android virus detectors (AVDs) in our study

ID	Vendor	AVD package name & version #	Downloads #	ID	Vendor	AVD package name & version #	Downloads #
1	Avast	com.avast.android;.........3.0.6915	50M-100M	2	AntiyAVL	com.antiy.avlpro;................2.3.02	50K-100K
3	AVG	com.antivirus;.........................3.6	100M-150M	4	AegisLab	com.aegislab.sd3prj.av;.........2.1.5	100K-500K
5	Avira	com.avira.android;.................3.1	1M-5M	6	Qihoo360	com.qihoo360.mobilesafe;....4.0.1	150M-200M
7	Bitdefender	com.bitdefender.security;2.8.217	1M-5M	8	Bornaria	com.bornaria.antivirus;.......1.3.60	50K-100K
9	Kaspersky	com.kms (premium);.........11.2.3	5M-10M	10	F-Secure	com.fsecure.ms.dc;........9.0.14504	100K-500K
11	ESET	com.eset.ems2.gp;..........2.0.843	1M-5M	12	CleanMaster	com.cleanmaster.security;......5.1.0	100M-150M
13	Dr. Web	com.drweb.pro;...............7.00.11	10M-50M	14	Ikarus	com.ikarus.mobile.security 1.7.20	100K-500K
15	Lookout	com.lookout;.........8.28-879ce69	50M-100M	16	Comodo	com.comodo.pimsecure;.......2.4.5	100K-500K
17	McAfee	com.wsandroid.suite;...4.0.0.143	5M-10M	18	CMCInfoSec	com.cmcinfosec.mbsec;3.11.18/u5	100K-500K
19	Norton	com.symantec.mbsec;....3.8.0.12	10M-50M	20	NetQin	com.nqmobile.antivirus20;..7.0.10	150M-200M
21	TrendMicro	com.trendmicro.tmmspsnl;....5.0	1M-5M	22	Sophos	com.sophos.smsec;..........3.0.1244	100K-500K
23	Tencent	com.tencent.qqpimsecure;.....4.6	1M-5M	24	Panda	com.pandasecurity.pandaav;.....1.1	100K-500K
25	Webroot	com.webroot.security;3.6.0.6579	500K-1M	26	AnGuanJia	com.anguanjia.safe;..............4.5.2	100K-500K
27	G Data	de.gdata.mobilesecurity;...24.5.4	500K-1M	28	Virusfighter	com.virusfighter.android;.......2.1.3	500K-1M
29	TrustGo	com.trustgo.mb.sec;.........1.3.13	5M-10M	30	Zoner	com.zoner.android.av;...........1.8.3	1M-5M

ables the AVD developers to come up with an ad-hoc design to perform *continuous* monitoring. However, our further detailed measurement study shows that these four AVDs only monitor several selected folders (e.g., the file *Downloads* folder) and only focus on limited file types (e.g., Android Application Package (APK) file). Therefore, by dropping the malicious payload onto any non-monitored folders or simply using an archived version of a malicious payload, *light monitoring malScan* can be evaded. This analysis indicates that the *FileObserver* based *light monitoring malScan* lacks *comprehensiveness*.

On the other hand, all 30 tested AVDs have designed the *heavy sweeping malScan* operation, which is a *comprehensive* malware scan. However, it is just due to the comprehensiveness of this type of scan that distinguishable system resource (e.g., CPU and memory) usage patterns are emitted when scanning. Also, we find that Android records the per process/per thread resource usage statistics in $/proc/[PID]/stat$ and $/proc/[PID]/task/[TID]/stat$ respectively, thus the resource usage patterns of each AVD's process/threads are readily available for all unprivileged third party apps. As a result, the scanning status of *heavy sweeping malScan* can be revealed by fingerprinting the high peaks of the AVD's resource usages, which makes the scanning vulnerable to targeted evasions. Thus, adversaries can evade this scan by identifying its scanning period and perform malicious action subsequently. So to clearly differentiate their scanning and idle periods and provide proof-of-concept evasions, we leverage *Fast Fourier Transformation* (FFT) to preprocess the per process/thread usage signals. We design and implement a *signal steganography* technique to identify the scanning locations (folders). We demonstrate that the adversary can plan targeted evasions by recognizing the (sub)folders that have just been scanned. Our analysis shed the light on the importance of developing a malware scanning mechanism that has both the *comprehensive* and *continuous* scanning properties.

Protection Assumption II: We then further assume AVD keeps monitor the whole file system completely and efficiently, and it can even perform behavior-based detections efficiently. By adding such a strong protection, we aim to find if there are still security holes in deploying AVDs on Android system.

Hazards in engineUpdate. Our study and various sources [23, 10] also show that AVDs on the fast-evolving Android system tend to perform engine updates (*engineUpdate*) fairly frequently, since AVD *engineUpdates* serve for various important tasks, including *malScan* mechanism enhancement, vulnerabilities [8, 20, 21, 19] patching and other functionality improvement [7]. However, we discover that the critical *AVD engineUpdate* procedure itself can cause devastating hazards to AVD even under both *Protection Assumption I and II*. In Android, Package Manager Service (PMS) updates the engine by removing the whole APK file and killing the AVD processes. Various components in the system have to perform complicated tasks (e.g., Dalvik bytecode optimization, configuration file parsing and etc.) for the newly downloaded APK file before setting up the AVD engine and reactivating it. As any strong defense mechanism relies on continuous running processes of the AVDs, this seemingly safe but complicated design of Android app-updating mechanism surprisingly kills the AVD process and nullifies any *perfect* protections for a period of high risk, which is called *null-protection window*. What's worse, our analysis shows that some AVD developers leave the engines inactivated after finishing its update. The lack of consideration in the AVDs' design indicates that AVD developers have not realized the potential hazard in *engineUpdate* yet. Without the automatic relaunch functionality, users can be exposed under high risk for a longer period.

The main contributions of this paper are summarized as follows:

- *New Understanding and New Hazards.* We reverse-engineer 30 widely-deployed Android AVDs and build a framework to conduct the first empirical study towards a comprehensive understanding of their design logic. Our study discover new hazards in *malScan*, *engineUpdate* and other mission critical operations of Android AVDs.

- *Measurements and Implementation.* We fully measured various types of *malScan* mechanism on Android and then design and develop targeted evasions against antivirus based on FFT and *signal steganography* techniques. For the hazards in *engineUpdate*, besides measuring the length of the *null protection window* on various hardware devices, we identify and analyze the Android AOSP source code for the program logic that creates the vulnerable period. We then built a *model checker* to automatically verify the existence of the discovered hazard in all Google Android versions from v1.5–v4.4.4, as well as vendor customized OS images.

- *Industrial Impact.* Because of the seriousness of these hazards and the great impact of current Android AVDs (around one *billion* user downloads), we immediately reported our findings to the antivirus vendors across 16 countries, and proposed mitigations designs to them. Also, not only can this *engineUpdate* hazard hurt the AVD apps, but it will impact other Android apps that require continuous running/monitoring (e.g. Mobile Device Management apps (MDMs) or Intrusion Prevention Systems (IPSs)) as well. Hence, we reported the hazard and proposed feature enhancements to the security team at Google, and they responded immediately and confirmed our findings.

2. BACKGROUND

2.1 AVD Background

Android is an operating system built upon the Linux Kernel. On top of the Linux kernel, Android is loaded with four software layers, namely *System Libraries, Android Runtime, Application Framework* and *Application.* In addition to the native Linux basic access control mechanism, Android provides a fine-grained permission mechanism for all the apps running on the Application layer, including all the AVDs from third party vendors.

Generally, Android uses a standard template process, the Zygote, for all the Dalvik Virtual Machine (DVM) processes warm-up phase. Zygote is the parent process for all the Android DVM processes, including all the AVDs' main processes. Each AVD is assigned its own unique user ID (UID) at install time, and the access control bits for the relevant files and folders in the file system are then set accordingly by the system. The dedicated group ID (GID) numbers are assigned based on the requested permissions for system resources. For instance, if an AVD wants to perform scanning operations on the files in the *sdcard*, it must first request the *READ_EXTERNAL_STORAGE* permission. Then, the system adds a relevant GID number (1028) for it and the process is put into the *AID_SDCARD_R* group to enable its access to the *sdcard.*

In the current Android system, a Binder interprocess communication (IPC) mechanism is used for more efficient *Intents* based message passing. Intents can be used in different purposes. For the app relevant usages, it can be sent to start an activity, start a service, deliver/receive a broadcast and so on. The system delivers various broadcast intents for system events to the apps/services that have registered to it. An AVD app/system daemon usually listens to specific broadcast intents by registering a broadcast receiver in the file AndroidManifest.xml or programmatically registering the relevant receiver in the code. For example, if an AVD app wants to relaunch automatically after the system boots up, it has to register for the system-broadcasted intent *BOOT_COMPLETED*, which is issued by the system once the boot process is completed. Some AVDs perform scanning for newly installed apps through registering the *PACKAGE_ADDED* events generated from the Package Manager Service (PMS) component. Listening system-broadcasted intents enables the AVD to keep track of system events of interest that are happening and then take appropriate actions.

2.2 AVD Behavior Analyses

Table 1 lists 30 popular AVDs that we selected from Google Play in Feb. 2014, which enjoy a total user downloads of about one billion. We pick the top 30 AVDs based on their overall protection rankings, according to AV Test Reports [6] for the period of Jan. 2014–April. 2014. To better understand the internal design of virus detectors on the Android platform, we further build a framework to perform comprehensive analyses of the selected AVDs.

2.2.1 *AVD Behavior Analysis Framework*

Our framework, illustrated in Figure 1, includes *Dynamic Tester*, *Static Code Analyzer* and *Environment Information Collector* to collect relevant virus detectors deployment information from both Android framework and Linux layer of the system.

The *Dynamic Tester* is built to interact with the AVDs, so that we can repetitively test some interesting properties of AVDs' runtime behavior. We wrote python and shell scripts to glue together the ADB tool, the Monkey-Runner tool [4], and the DDMS tool [3] from Android Studio. Monkey-Runner is a testing program that can send user and system events for our testing purpose. We leverage DDMS, a common debugger for Android developers, to collect relevant execution traces for particular operations for further analysis.

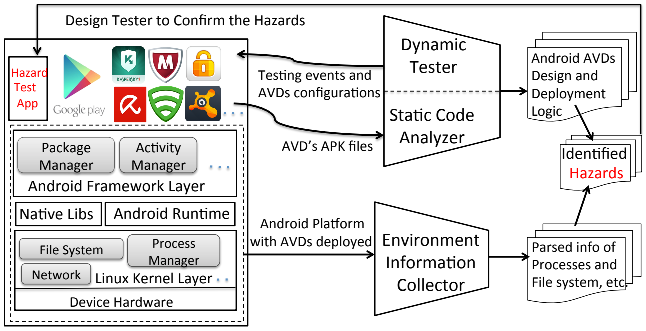


Figure 1: The AVD testing framework for Android platform

In the *Static Code Analyzer*, we first decompile the APKs of AVDs and then conduct analyses on the required *permissions* and registered *intent actions* in the corresponding broadcast receiver components. To further understand the design logic of AVDs, we build the control flow graph (CFG) upon the Android Dalvik bytecode disassembled from *baksmali* tool [16]. We adapt some of the analysis modules from Androguard [1] to construct the CFGs. We notice that most of the AVDs are not highly obfuscated, so the reverse engineering process is comparatively straightforward.

We also build an *Environment Information Collector* to analyze both the Android framework layer and the underneath Linux layer with all the AVDs installed, configured and running on the system. The purpose is to discover some effective public *information leakage channels*, which can help any third party app infer the AVDs' running status. For the Linux layer, we parse the information about the files and folders (e.g., access control bits) in the file system and the information about the running processes (e.g., the UID and GID numbers and their forking orders). Then we focus particularly on analyzing the information that is relevant to the installed AVDs. For the Android framework layer, we analyze relevant Android APIs to collect information about AVDs running status. After several rounds of automated testing and manual result verification, we discover several interesting *information leakage channels*, shown in Table 2, which will be especially valuable in determining the vulnerable periods of the running AVDs. Note that none of the discovered channels are protected by any Android permission yet.

2.2.2 *Intents Registered and Permissions Claimed*

In Table 3, we list the types and frequencies of the registered *intent actions* and claimed *permissions* of the 30 AVDs collected based on our reverse engineering result. Our analysis result indicates that current AVDs register lots of interesting permissions for privileged operations; for instance, the *KILL_PROCESSES* permission is used to kill suspicious background processes and the *ALERT_WINDOW* permission is claimed to help pop up urgent alert window from the system when suspicious status is identified. We also see that the current AVDs emphasize a *continuous* running status on the Android system. For instance, all these AVDs listen to the *BOOT_COMPLETED* system event to provide complete protection after the system boots up. They also obtain the *WAKE_LOCK* permission to periodically wake up the CPU to keep monitoring the system status. Events like *PACKAGE_ADDED* and *PACKAGE_REMOVED* are mostly registered to help monitor the newly installed/updated Android application package (APK) files. However, we still identify a loophole in its *engineUpdate* operation that unexpectedly nullifies the AVDs' protection.

Table 2: Information leakage channels used to infer AVDs' running status

ID	Probing Channels	Explanation
1	ActivityManager.getRunningAppProcesses()	Leaks DVM processes meta info (e.g., UID, PID, process name and etc.)
3	ActivityManager.getProcessMemoryInfo()	Leaks DVM process memory info (e.g., nativePSS, dalvikPrivateDirty and etc.)
4	PackageManager.getInstalledApplications()	Return a List of all the packages of installed applications
5	PackageManager.getPackagesForUid()	Leaks the names of all packages that are associated with a particular UID
6	PackageManager.sendBroadcast(Pkg_add)	Leaks the AVD or other apps' installation status
7	PackageManager.sendBroadcast(Pkg_rm)	Leaks the AVD or other apps' removal status
8	PackageManager.sendBroadcast(Pkg_updt)	Leaks the AVD or other apps' update status
9	Build.MODEL (HARDWARE)	Leaks device type and hardware specifications information
10	/proc/[AVD_pid]/stat	Leaks the relevant CPU usage info (e.g. utime, stime and etc.
11	/proc/[AVD_pid]/task/[AVD_tid]/stat	Leaks tasks CPU usage info of AVD process, used to purify the fingerprinting result
12	/proc/[AVD_pid]/statm	Leaks the memory usage info (e.g. the RSS, virtual memory size, etc.)
13	/proc/[AVD_pid]/status	Leaks the memory usage info (e.g., the RSS counts in pages)
14	JNI call stat() /data/data/[AVD_package]/*	Leaks all the files' meta info (e.g., size, creation date and etc.) in the subfolders
15	/system/bin/ps	Leaks all the running processes info including the DVM and non-DVM processes.
16	/proc/uid_stat/[AVD_uid]/snd(rcv)	Leaks the network usage info (e.g., send and received package sizes, etc.)

Table 3: Intents registered and permissions asked by AVDs

Intents Registered	#	Permissions Requested	#
MEDIA_REMOVED	4	SUPERUSER	1
MEDIA_CHECKING	5	BATTERY_STATS	12
PWR_DISCONNECTED	5	google.c2dm.RECEIVE	17
WIFI_STATE_CHANGED	6	KILL_PROCESSES	21
DATE_CHANGED	6	COARSE_LOCATION	21
SERVICE_STATE	7	ALERT_WINDOW	22
DIAL	7	WRITE_BOOKMARKS	22
MEDIA_UNMOUNTED	8	GET_ACCOUNTS	22
POWER_CONNECTED	8	READ_SMS	23
net.wifi.STATE_CHANGE	10	READ_BOOKMARKS	23
MEDIA_EJECT	10	READ_CONTACTS	25
USER_PRESENT	10	RECEIVE_SMS	25
ACTION_SHUTDOWN	12	SEND_SMS	25
NEW_OUTGOING-CALL	15	READ_LOGS	26
PHONE_STATE	17	GET_TASKS	26
PACKAGE_REPLACED	23	WAKE_LOCK	29
PACKAGE_REMOVED	30	EXTERNAL_STORAGE	30
PACKAGE_ADDED	30	READ_PHONE_STATE	30
BOOT_COMPLETED	30	BOOT_COMPLETED	30

Road map: The remainder of the paper continues as follows: Section 3 elaborates hazards relevant to the *malScan* mechanism, followed by the algorithm designed and implemented to confirm the hazards. Section 4 explains the exploring process of the hazard in AVD *engineUpdate* procedure and a model checker is developed to confirm the relevant vulnerable program logic. Other hazards are discussed in Section 5, followed by proposed mitigations. Section 6 reviews related works and Section 7 concludes this paper.

3. HAZARDS IN MALWARE SCAN

In this section, we report our study on the two malware scan (*malScan*) operations, namely the *light monitoring malScan* and the *heavy sweeping malScan*, and then followed by the discovered hazards for each type of *malScan* operations and their corresponding measurement results.

3.1 Malware Scan Behavior

Different from previous works on antivirus security measurement [44, 40] that targeting the weakness of the incomplete signature database, we assume that AVDs are all equipped with a *complete* VDF, which includes all known malicious payloads and their obfuscated counterparts. Hence, we use the easily recognizable malicious payloads from the well-known Malware Gnome Project [53] in our testing. This will force us to concentrate on analyzing the *malScan* mechanism itself when deployed on the Android platform. Similar to PC AVDs, the *malScan* operations are one of the core functionalities of AVDs for malware recognition based on up-to-date VDF.

Our analysis result shows that only four of the tested 30 AVDs have developed *light monitoring malScan*, which performs short-term scans for particular file system status changes. It leverages the *FileObserver* APIs provided by the Android framework, which is based on the *inotify()* system call to monitor the file system changes (e.g., create, modify, delete files/folders and etc.). For example, when a new file is downloaded into the $/sdcard/Downloads$, the AVD will perform a very quick scan for that particular file system status change on the folders, given that the particular folder is specified to be monitored in one of the registered FileObserver instances. However, our further analysis indicates that it can only focus on a few pre-selected folders (e.g., $/sdcard$) and the scan is lightweight, which means it cannot handle archive format.

On the other hand, all the 30 tested AVDs have implemented the *heavy sweeping malScan*, a comprehensive scan on the file system, which can be pre-scheduled or triggered directly by a user. It is usually performed by one or more Android service(s) or dedicated native process(es) (AVD # 9). This scan will consider most files, including archive files (e.g., *.apk*, *.tar* and *.zip*), in the target folder and its (sub)directories. It then fingerprints them against the VDF diligently. For instance, when targeting the $/data/app$ folder, it will check all the compressed app packages, the Android Application Package (APK) files of all installed apps, and iteratively scan through subdirectories and files in each APK file package. If an APK file happens to contain an archive file format, it will be uncompressed and scanned through all files contained in the archive file. Therefore, all of the tested AVDs rely on *heavy sweeping malScan* to perform comprehensive malware detection.

3.2 Light Monitoring malScan Hazard

To discover the potential hazards in *light monitoring malScans*, which leverage the *FileObserver* APIs, we use the *Dynamic Tester* to fully measure the effectiveness of the four AVDs. Our *Dynamic Tester* drops well-known malicious payloads on the $/sdcard$, which are recognized directly by all the four AVDs that have implemented this type of *malScan*. Then we configure the *Dynamic Tester* to test different dropping folders and using various archiving formats

Table 4: Light monitoring malware scan ineffectiveness

Tested Malicious Dropping Actions	Reacted AVD #
APK files (zipped) dropped on /sdcard	14
APK files (zipped) dropped on /sdcard/*/	None !!!
Native files (zipped) dropped on /sdcard	14, 15
Native files (zipped) dropped on /sdcard/*/	None !!!
APK files dropped on /sdcard/	13, 14, 15, 22(17.2s)
APK files dropped on /sdcard/*/	13, 15, 22
Native files dropped on /sdcard/	13, 14, 15
Native files dropped on /sdcard/*/	13

(e.g., zipped files). Table 4 contains the reacted AVDs for each specific measurement case. The result demonstrates that this type of scan is not effective, especially for zipped payloads dropped in a deeper-level folder of the targeted folder (e.g., $/sdcard/*/$). One reason that current AVDs cannot implement recursively file system monitoring due to the fact that this *FileObserver* APIs are not designed for comprehensive file system hooking.

From the result, we can also tell that AVD #15 from Lookout Inc., which is one of the top mobile security company targeting Android security for years, provides the best design for the *light monitoring malScan* among the tested AVDs. However, a design vulnerability in the AVD #15 has been discovered from our detailed analysis. The *light monitoring malScan* functionality is accidentally blocked when performing the *heavy sweeping malScan*. This buggy case indicates the problematic design of *light monitoring malScan* based on *FileObserver* APIs. Besides this problem, although our measurement shows that the AVD #22 Sophos can recognize malicious payloads in the $/sdcard$ directory, on average it takes 22.7s to react to the observed payload dropping events reported by the *FileObserver*. This vulnerability is due to an inappropriate way to handle the triggered events from the FileObserver object. During such a long reaction period, the adversary can easily finish his malicious actions based on the dropped payload and remove it before the *malScan* (its actual malware recognition action). We have reported the vulnerabilities to Lookout and Sophos respectively, and they have planned to fix them in the next versions.

Since *FileObserver* APIs provided by Google is not meant for AVD scanning purposes, these APIs cannot cover the whole file system comprehensively and efficiently. Thus, the ad-hoc design of the *light monitoring malScans* based on FileObserver is proven to be ineffective and problematic. Therefore, we are calling the need for specific hooking APIs for AVDs *light monitoring malScan* from platform designers, for instance, Samsung, Google and so on.

3.3 Heavy Sweeping malScan Hazard

Different from the *light monitoring malScan*, the *heavy sweeping malScan* is comprehensive, so it is usually hard to bypass. Especially, when the AVDs are equipped with the *complete* VDF. The *heavy sweeping malScan* is a series of expensive operations performed in a period, we call the period *high protection window*. Since it can be only triggered implicitly by a pre-defined schedule or explicitly by a user, this type of *malScan* can be very *comprehensive*, but lack of *continuous* protection. Therefore, if the adversary can identify the *high protection window*, it can potentially evade the detection by performing malicious actions after the scan.

To identify the *high protection window*, we want to analyze if the malScan reveals any side-effects, which is different from other non-scan operations. Based on the observations of *malScan* of the 30 AVDs, we notice that it is exactly this *comprehensive* scanning strategy that produces very distinguishable CPU and memory usage patterns. These patterns are distinct from other AVD operations (e.g., VDF updates or other managing tasks of AVDs). The result from the *Environment Information Collector* in Table 2 shows that one can directly access the resource usage pattern (e.g., the utime and stime[1]) of AVDs from the file $/proc/[AVD\_pid]/stat$, which is accessible by the public. Here the *AVD_pid* is the process id of the AVD and the mapping between the AVD package name and its *pid* can be obtained through the probing channel #1 and #15.

Table 5: Heavy-sweeping malware scan configurations

Type	AVD #
Full Scan Only	5, 6, 7, 11, 15, 17, 21, 22, 25, 26, 29
Only Quick/Full	8, 9, 13, 14, 20, 24, 23, 27
App Only	1, 2, 4, 7, 12, 16, 17, 19, 20, 21, 27-30
Folder Only	1, 2, 3, 4, 9, 12, 13, 17, 18, 28, 30
Pre-scheduled	1, 2, 3, 4, 7, 9, 12, 16, 17, 19, 20, 21, 27, 29, 30

Table 6: Hardware specifications of four Android devices

Devices Name	OS versions	RAM	CPU cores,speed
Google Nexus 4	4.3 JB	2G	Quad,1.5kMHz
Asus Nexus 7 Tablet	4.1 JB	1G	Quad,1.3kMHz
Samsung Note 2	4.0 ICS	2G	Quad,1.6kMHz
Samsung Nexus S	2.2 Froyo	.5G	Single,1kMHz

Our further investigation shows that this Linux statistics file (*stat*), is created for the whole AVD process. Due to the multi-threaded programming of Android apps, the *stat* file for AVD process includes noise from other types of AVD operations, for example, a user interacting with the AVD app or other process management tasks (e.g. garbage collection) that are performed simultaneously on the background. To precisely identify the scanning operations and fingerprint the scanning period and locations, we discover that one can directly leverage the task information in the subdirectory of each $/proc/[AVD\_pid]/$ folder, namely the $/proc/[AVD\_pid]/task/[AVD\_tid]/stat$, #10 and #11 side channels. They reveal the utime and stime of a particular scanning task of an AVD process (e.g., start and end of the *heavy sweeping malScan*). Several dedicated tasks (main user interface or garbage collection threads) are assigned with fixed *tid* based on the *pid* offset. Thus, the noise introduced from these tasks or operations can be filtered out.

Another interesting observation is that some AVDs (e.g. # 9) use dedicated native process(es) for more efficient scanning, but this special improvement in *malScan* surprisingly hurts itself, as it simplifies the relevant *high protection window* fingerprinting procedure. One only has to identify the *pid* of the "/kav/libscan" native process of #9 through the *information leakage channels* #1 and #15. And then the relevant resource usage patterns for the comprehensive scanning tasks can be further collected through *channels* #10 and #11 of the "/kav/libscan" process.

Testing malScans and Collecting Scanning Traces.

To further explore the feasibility of fingerprinting the *heavy sweeping malScan* in real time and build proof-of-concept evasions from the collected system resource (e.g., CPU and memory) usages of the targeted AVD, we first run dynamic testing over all the AVDs' *heavy sweeping malScan*. The result of our dynamic testing, in Table 5, shows that current AVDs have three types of configurations for this *malScans*: 1) only one full scan; 2) quick/full scan, where a quick scan usually scans a subset of full scan's content (e.g., only the APK files); 3) flexible scan by selecting different targets (folders). In addition to the above, some AVDs also have user-scheduled scans at a particular time of a day or in a weekly/monthly basis. To perform one *malScan*, it usually takes 1-10 minutes, depending on the number and type of files/apps to be scanned. To build a bet-

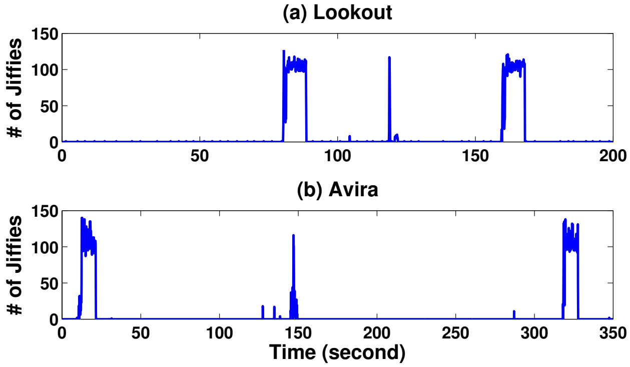


Figure 2: 30 minutes CPU usage (utime) temporal statistics of (a) AVD Lookout and (b) AVD Avira

ter fingerprinting algorithm, we collect the resource usage statistics for various *heavy sweeping malScan* configurations of all the 30 tested AVDs on four devices with different hardware specifications, as shown in Table 6. For each AVD on each device, we perform ten 30-minute tests to collect its corresponding resource usage statistics. We name the resource (e.g., CPU and memory) usage statistics for a particular job of an AVD as a *trace*. A *trace* generally contains all the relevant CPU and memory information (e.g., the utime, stime[1] of CPU and the VmSize, VmRSS[2] of the memory). For a particular AVD scan trace, we name each specific resource usage dynamics as a *signal*. For instance, the process's *utime signal* is a series of incremental values calculated based on the collected user mode CPU usage time (in Jiffies), which reflects the AVD's real-time CPU usage. Various signals can be sampled at adjustable rates (e.g., 1 sample/second or 5 sample/Minute).

Identify Heavy Sweeping malScans Traces.

Our first goal is to differentiate the traces of *heavy sweeping malScan* with traces from all other operations (e.g., VDF file updates, cloud Scan and etc.). Figure 2 contains 30-minute time series of two representative AVDs (AVD Lookout and Avira) during which the user triggers two *malScans* and various other random AVD operations. We remove part of the blank sub-period that contains no system resource statistics. We observe that the utime pattern (stime has similar pattern) has an extremely aggressive CPU usage pattern in a continuous time period, which is the *high-protection window*, compared to all other small traces introduced by non-scan operations. Therefore, we use the average *utime, stime signals* in a sliding time window (around 30 seconds) to decide if the AVD's scan enters the *high-protection window*.

Differentiate Full and Quick Scans.

Besides identifying the *high-protection window*, we want to further differentiate the *high-protection window* of full and quick scans of one particular AVD, so that the adversary can start malicious actions (e.g., dropping/executing malicious payloads) right after the more comprehensive full scan, which is safer to evade detection. Figure 3 contains the *stime signal* of representative full and quick scans of two AVDs, from which we can tell that the quick scan is generally shorter (quicker) than full scan, and the stime signals' (utime is the same) peak and average values are pretty close to full scan. Based on this general heuristic, we can differentiate full and quick scan by collecting two *malScans* with very different signal lengths as benchmarks. Therefore, when the next *malScan* is identified, its signal length can be compared with the previously stored benchmarks. We also keep updating the corresponding benchmarks with the newly classified scan traces for different categories.

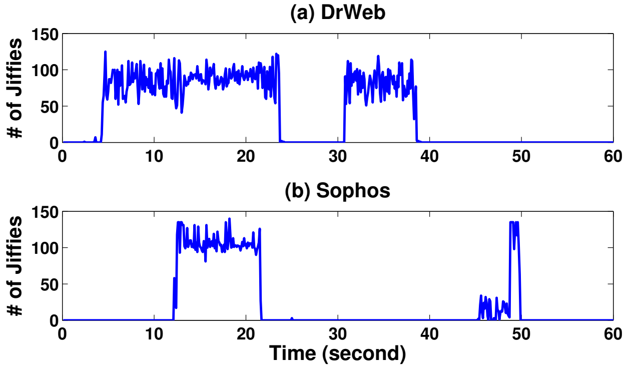


Figure 3: Comparison of full/quick scans' stime signal of (a) AVD Dr.Web and (b) AVD Sophos

Signal Steganography based Scan Location Inference.

Our study also shows that some AVDs provide multiple ways of configuring the *heavy sweeping malScans* that target on various scanning locations. On rooted devices, some AVDs [12, 14] even enables the ability to scan most the important folders (e.g., $/sdcard$, $/data/app$, $/data/data$, $/system/app$ and etc.). Therefore, we seek to identify the precise location that the scanning has just happened. Because it is less likely for the AVD to re-scan the same location within a short period of time, one can design an evasion attack based on this assumption.

The scanning location inference idea is very intuitive. The goal is to infer if the scan is just sweeping through a particular folder of interest. To achieve this, we leave a special file, namely a *marker*, in that folder. The *marker* file is intentionally designed to exhibit special patterns in terms of CPU or other resource usages during AVD *heavy sweeping malScan*. Since only the *marker* file designer knows the embedded signal and later extracts it. We call this method, *signal steganography*.

Delay-of-Scan Hazard. The design of a unique marker file is based on our observations of the normal AVD scanning process. First, most of the AVDs have aggressive CPU usage, mainly caused by checking the files in the target folder. The pattern of CPU utilization during scanning varies depending on the type, size and structure of the (archive) files. Second, once encountering an archive file, most AVDs would have to first unzip the file before fingerprints it. The unzipping operations lead to less CPU utilization but more memory use, which exhibits certain patterns (the shape and frequency of CPU or memory utilization profiles). If one could make this special pattern continue for a significant long time (e.g., several seconds, which is actually unnoticeable compared to the minute-level scanning time), we can identify and extract it directly from the whole trace and finish the discovery of the embedded *signal*. To leverage these observations, we create a special archive file

[1] utime is the user mode jiffies and stime is the kernel mode jiffies, which are the 14th and 15th field of the /proc/AVD_pid/stat file, respectively. Jiffies is a time measure based on a clock maintained by the kernel. CPU time, which is limited by the resolution of the software clock and measured by clock ticks (divided by sysconf(_SC_CLK_TCK). _SC_CLK_TCK is set to 100 in bionic/libc/unistd/sysconf.c for all Android OS versions.

[2] For an AVD process, the VmSize is the virtual memory size and VmRSS is the portion of a process' memory that is held in RAM, which are the 1st and 2nd fields of the s/proc/[AVD_pid]/statm, respectively.

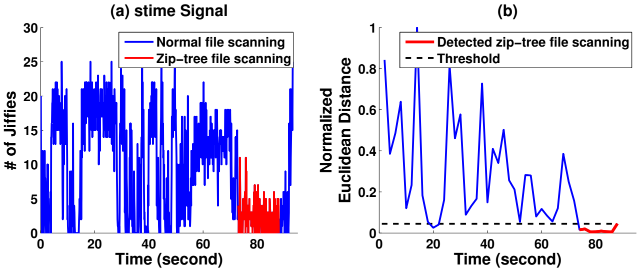


Figure 4: Stime signal (red line indicates zipTree scanning segment) of Panda Security

as the *marker*, which is a three-layer zip file called *zipTree*. Each parent node in this *zipTree* entry contains 10 zip files, and each one of the 10 zip files contains another 10 child-zip files. The inmost *leaf* zip file is an empty *.txt* file. The crafted *zipTree* is about 0.7KB in size, which can hold the *malScan* process for 9 seconds. We name this *Delay-of-Scan* hazard, which can potentially cause denial of service attack given a big enough *zipTree*. However, in our *signal steganography* based pattern recognition approach, we only need to delay the scan for a couple of seconds. In this way, we can precisely identify the scanning locations based on the marker that was dropped on the particular folder.

Figure 4(a) displays the stime signal of the scanning by Panda Security. It is noteworthy that the stime signal during *zipTree* scanning (as shown in a red line in the figure) exhibits patterns with three special *properties* compared to other parts of the signal: (1) the signal has a much lower mean value than other parts of the stime signal; (2) the signal shows stable patterns; that is, the signal is periodic and has relatively deterministic frequencies; (3) the stable pattern of the signal of *zipTree* file scanning lasts for considerably longer time compared with the other part in the signal. Based on these observations, we devise an efficient online algorithm, Algorithm 1, to recognize/extract the stime signal segments of *zipTree*. The algorithm is based on *Fast Fourier Transformation* (FFT) for the classic spectrum analysis that is widely used in signal processing. FFT transforms the sample signal from the time domain to the frequency domain, so our analysis is stable against potential time variations in the signal. More importantly, the frequency domain can still capture most part of the three properties of the signal. The idea is to determine *zipTree* scanning period by identifying a period of time that exhibits similar CPU utilization patterns (e.g., similar means and magnitude in different frequencies), or in one word, similar frequency spectrum characteristics that appeared repetitively. Once we identify a period with lower mean value signals (based on property (1)), we first perform FFT (line 9 in the Algorithm 1) to multiple sliding windows of the CPU utilization signal and generate a time series of frequency spectrum, a presentation of a time-series signal in the frequency domain. As is well known in signal processing, similar two frequency spectrum imply that the corresponding two time series sharing similar signal components at all frequencies. We then compute the *Euclidean Distance* of the neighboring spectrum (line 10). We determine two neighbor frequency spectrum as ”similar” by checking if the euclidean distance between these spectrum is smaller than a self-adapted threshold (line 11). Since the archive files produce property (2) (3), the distance can be kept very small for an amount of time (several seconds, controlled by causing the *Delay-of-Scan* hazard), which is demonstrated in the red part of Figure 4(b). Although the stime signal length caused by the *Delay-of-Scan* hazard varies on different hardware devices, the repeated stable structure in the *zipTree* file ensures the periodicity of the low signal pattern regardless of hardware specifications. Note that for all the *malScan* inference measurements, we reach the desired results by using only the stime signals from the CPU patterns.

Algorithm 1 Pseudo-code for the algorithm to extract zipTree scanning pattern

```
 1: $zipfilescaning \leftarrow$ False
 2: $window$[window_size] $\leftarrow$ empty
 3: $spectrogram$[window_size][ ] $\leftarrow$ empty
 4: k $\leftarrow$ 0, $mag\_thresh \leftarrow \alpha$
 5: $dist\_thresh \leftarrow \beta$, $len\_thresh \leftarrow \gamma$
 6: while obtain new value $V$ do
 7:   update_Window($window$, $V$)
 8:   if mean($window$) $< mag\_thresh$ then
 9:     $spectrogram[k][] \leftarrow$ FFT($window$)
10:     $distance \leftarrow$ Euclidea_Distance($spectrogram$[k], $spectrogram$[k-1])
11:     if $distance < dist\_thresh$ and $k > len\_thresh$ then
12:       $zipfilescaning \leftarrow$ True
13:     else
14:       $zipfilescaning \leftarrow$ False
15:       erase($spectrogram$)
16:       k $\leftarrow$ 0
17:     end if
18:   else
19:     $zipfilescaning \leftarrow$ False
20:     erase($spectrogram$)
21:     k $\leftarrow$ 0
22:   end if
23: end while
```

3.4 Targeted Evasions

Based on the discovered hazards, various targeted attacks can be designed. One potential attack scenario is based on repackaging [49, 32, 26, 48, 34] a benign carrier (RBC app) app, which does not contain any malicious payload for root exploiting, or privacy stealing. Therefore, the RBC app itself will not be judged as malicious during the install-time scan. After the RBC app is deployed on the system, it can leverage the *information leakage channel* #1 in Table 2 to check the name of the installed AVD and use the discovered hazard to infer its scanning status. Once it identifies a moment to its advantage (e.g., end of a *malScan*), malicious actions can be performed under the radar of the AVD's scans. For example, based on the similar analysis result from Table 4, one can drop malicious payloads to the folder that is not covered by the *light monitoring malScan*, load and execute them and then remove the evidence. Other remote evasions can also be designed. For instance, one can leverage remote vulnerability exploiting (e.g., code injection vulnerability in HTML5-based apps [37]) to hijack a running process's control flow, and then conduct the similar evasions to prevent the injected payloads from being identified.

4. HAZARDS IN ENGINE UPDATE

For an AVD equipped with complete VDF and perfect hooking mechanism, we consider the best moments to evade it is when its process(es) has not been activated yet. Our study result in Table 3 shows that all the tested AVD are listening on the *BOOT_COMPLETE* intent to automatically launch itself when the system boots up. This close a potential opportunity for evasions and indicates that AVD developers are aware of some similar hazards. However, after further analysis on the interaction between the AVD and the Android system, we identify a hazard in AVD's engine update (*engineUpdate*) procedure, that unexpectedly nullifies any strong protections an AVD provides for the system. Here we elaborate the AVD's *engineUpdate* operation in Android and analyze the hazard.

4.1 Engine Update Behavior and Hazard

The AVD *engineUpdate* operation provides essential updates for AVD *malScan* operation improvement, bug and vulnerability patching [7, 8, 20, 21, 19], and update-to-date protection [23]. Since the technical details of AVD's *engineUpdate* mechanism on Android have not been scrutinized before, to come up with a detailed full picture, we leverage our analysis framework and manual analysis based on runtime logs and related decompile Dalvik bytecode (the *smali* format). The *engineUpdate* operation is performed whenever an AVD developer posts an updated version of APK on Google Play. It can be either triggered by the user, or by the old AVD process automatically. The whole procedure is executed by the Android system, namely the Package Manager Service (PMS), the Activity Manager Service (AMS), *installd* daemon and etc. Lots of evidence [23, 10] show that AVDs update fairly frequently. Indeed, during a two-week period in March of 2014, we initially tested the process of *engineUpdate* operations using our analysis framework and noticed a high update ratio, 25 out of 30 tested AVD products have version updates from Google Play.

On the traditional PC platform, during the engine update procedure, an AVD will not shut down the whole AVD program, but only replace part of the necessary modules (e.g., DLL files or kernel drivers). However, we identify that on the Android platform, the updating procedure is different. One potential problem of this specific procedure is that the Android system will kill the AVD process(es) before the *engineUpdate*. This design is considered as valid and effective for most third party apps and help prevent unintended app update [15], but this specific design lacks detail analysis on potential security implications. Especially when considering from the perspective of AVD realtime protection and detection, this APK update mechanism introduces a serious flaw, as its complicated procedure renders the system lacking of AVD's protection for a period of high risk, which is called *null-protection window*.

In order to confirm the hazard, we further verify the existence of the *null-protection window* by checking the implementation logic in the Android AOSP code base. The relevant high-level program logic flow is generalized and presented in Figure 5 as a Finite State Automaton (FSA). In the device, the PMS from the Android framework layer listens on update requests and performs the actual work along with several other services. For instance, the *installd* daemon and the AMS have to collaborate with the PMS component to finish different tasks in the *engineUpdate* procedure. Based on the complicated design of app updates on Android, the system handles AVD version updates in several steps. After the new APK is downloaded and update is confirmed, the current AVD process is killed, and the old code directory ($/data/app$) and some relevant data files in the $/data/data/[AVD\_package]/*$ subdirectories are removed. Before relaunching the new process, the newly downloaded Android Package file (APK) of AVD is verified for the developer's signature, and followed by Dalvik bytecode optimization (e.g., *dex* to *odex*), configuration file parsing (e.g., permission registration based on the AndroidManifest.xml file) and etc. Note that the system never helps relaunch the previously killed process, which will expand the *null-protection window*. Due to this unavoidable *null-protection window* caused by the specific design for package updates of Android, the AVDs become vulnerable even under the strong *protection assumptions I and II*.

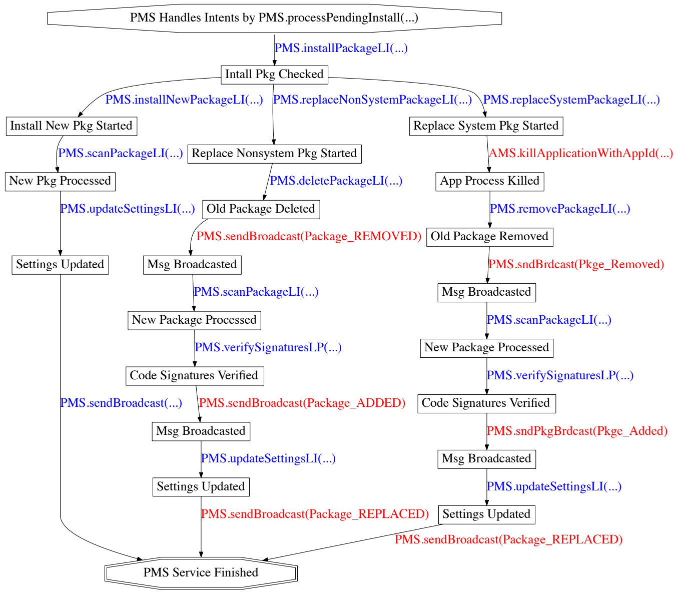

PMS Handles Intents by PMS.processPendingInstall(...)
PMS.installPackageLI(...)
Intall Pkg Checked
PMS.installNewPackageLI(...)
PMS.replaceNonSystemPackageLI(...)
PMS.replaceSystemPackageLI(...)
Install New Pkg Started
Replace Nonsystem Pkg Started
Replace System Pkg Started
PMS.scanPackageLI(...)
AMS.killApplicationWithAppId(...)
New Pkg Processed
PMS.deletePackageLI(...)
App Process Killed
PMS.updateSettingsLI(...)
Old Package Deleted
PMS.removePackageLI(...)
PMS.sendBroadcast(Package_REMOVED)
Old Package Removed
Settings Updated
Msg Broadcasted
PMS.sndBrdcast(Pkge_Removed)
PMS.scanPackageLI(...)
Msg Broadcasted
New Package Processed
PMS.scanPackageLI(...)
PMS.verifySignaturesLP(...)
New Package Processed
Code Signatures Verified
PMS.verifySignaturesLP(...)
PMS.sendBroadcast(...)
PMS.sendBroadcast(Package_ADDED)
Code Signatures Verified
Msg Broadcasted
PMS.sndPkgBrdcast(Pkge_Added)
PMS.updateSettingsLI(...)
Msg Broadcasted
Settings Updated
PMS.updateSettingsLI(...)
PMS.sendBroadcast(Package_REPLACED)
Settings Updated
PMS.sendBroadcast(Package_REPLACED)
PMS Service Finished

Figure 5: The constructed FSA of PMS for Android v4.4.4_r1

To leverage the exposed *null-protection window*, the adversary must infer if the AVD is performing the *engineUpdate* operation. During our runtime testing and code analysis, we find that PMS will broadcast a series of intents with different action fields, including *PACKAGE_REMOVED*, *PACKAGE_ADDED* and *PACKAGE_REPLACED* (red in Figure 5), so as to notify the other components to perform the corresponding tasks and keep them updated with the current app updating status. However, these broadcasted intents unexpectedly reveal the updating status of AVD to any unprivileged apps. So the malicious app can simply register these broadcast intents to get notified, and then start malicious operations during the exposed *null-protection window*. Our analysis framework also helps us identify several other leakage channels (e.g., channel #1, #4 and #14 in Table 2) that can infer this vulnerable status of AVD. This indicates simply restricting the receivers of the broadcasted intents cannot fully eliminate the hazard.

4.2 Null-protection Window Length

Next, we quantitatively measure the length of this *null-protection window* for various AVDs and devices. We conduct four groups of experiments based on four types of real devices, whose configurations are provided in Table 6 in Section 3. For each group, we try to test all the 30 AVDs' engine update operations and record their *null-protection window* lengths. For every AVD on each device, we test 10 times to obtain the mean value and relevant confidence interval. Our result indicates that the window lengths are quite stable; for example, the representative Symantec AVD has an average window length of 11.2 seconds, with a confidence interval [-0.13, 0.13] at the confidence level 95%. This is because of the routinely PMS updating logic and the atomic nature for most of the PMS method invocations during the execution. Due to space limits, we only show the mean values in Figure 4.2. Here, the x-axis is the average length of *null-protection window* in seconds, and the y-axis represents each AVD. For each AVD, to clearly compare the window lengths for four devices, we draw four horizontal bars in different colors. Longer bars are partly covered by shorter bars. We can see that for all AVDs, the *null-protection window* lengths in Samsung Note II and Google Nexus IV (LG) are the smallest, whereas Nexus S (Samsung) has the largest window, from about 9.2-16.8 seconds. Note that all the collected results are the theoretical lower bounds for the *null-protection window*, because in this test, our *Dynamic Testor* keeps sending launching events from *adb* tool during each tested AVD updating session. This guarantees the automatic AVD relaunch right after the AVD is updated.

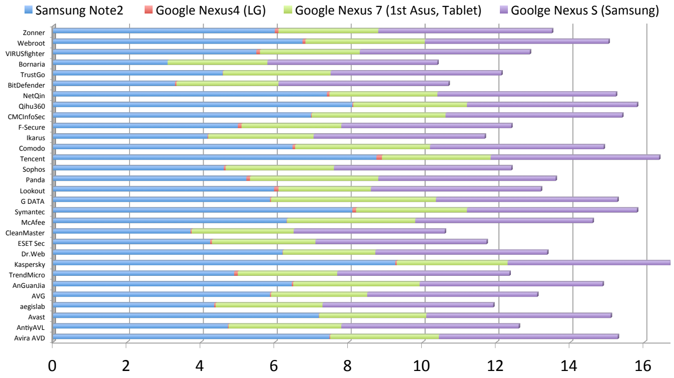
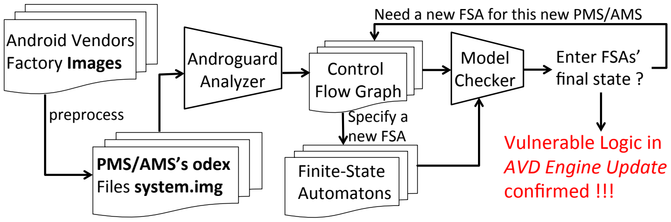


Figure 6: The *Null-protection Window* lengths (in seconds) for all 30 tested AVDs on four Android devices

Since we can automate the process of AVDs' relaunching, we think one might provide a preventive design to relaunch the AVD by the updated new AVD code itself immediately after the *engineUpdate* finishes. One interesting mitigation that we come up with is to add a *receiver* component in AVD to listen to the broadcasted *action.PACKAGE_REPLACED* intent, which indicates the finishing of the *engineUpdate*. Then this receiver of the updated new AVD engine is always triggered by this broadcasted intent right after the *engineUpdate* finishes. Within the registered receiver component, we then relaunch the main monitoring process automatically through Android APIs, *startActivity()*. Note that this mitigation does not solve the fundamental problem that the old AVD process is killed and the vulnerable period is created, but it can only reduce the *null-protection window* length to the theoretical lower bound (caused by the complicated updating operations carried out by the system). None of the 23 of the tested AVDs that registered the *action.PACKAGE_REPLACED* (in Table 3), are actually performing the preventive design for their own updates, but only use the registered receiver to start a service to scan other apps' updating for malware detection purpose. We report our findings to all the AVD vendors and propose this short-term mitigation to them. They responded to us immediately and confirmed the hazard. Similar patches will be provided in their next release based on our suggested mitigation. However, to completely eliminate the hazard, the PMS design should be enhanced. Since this hazard can be applicable to other apps that require continuous monitoring properties (e.g., Mobile Device Manager (MDM) [18], IPS [12, 14] and etc.), we reported this hazard to Google and they confirmed our findings and we are now working with them on possible feature enhancements for the relevant components and services.

4.3 Model Checking the Vulnerable Logic

To automate the process of verifying the pervasiveness of the vulnerable temporal logic in AVDs' engine update procedure, we leverage the model checking techniques, which have been applied widely [25, 29, 27] to perform system bugs and vulnerabilities examination and temporal property checking for both mobile and PC platforms. Our model checker for the Android platform is similar to Chen et al. [27], in the sense that we both check the reachability of certain temporal logic in the code. However, our model checker, particularly targets the Dalvik bytecode of the PMS in Android for the *engineUpdate*'s vulnerable temporal program logic.



Android Vendors Factory Images
preprocess
Androguard Analyzer
PMS/AMS's odex Files system.img
Control Flow Graph
Specify a new FSA
Finite-State Automatons
Model Checker
Need a new FSA for this new PMS/AMS
Enter FSAs' final state ?
Vulnerable Logic in AVD Engine Update confirmed !!!

Figure 7: The model checking framework

Figure 7 shows the design of our model checking framework. The inputs are the Android factory images collected from multiple vendors. After the preprocessing phase, we extract the relevant *.odex* file of the PMS components in the system.img. The odex code, namely the optimized Dalvik bytecode, can be run directly on the targeted Dalvik VM. We decompile the odex file into Dalvik bytecode and produce the CFGs for these components. From the constructed CFGs, we first identify the relevant execution traces that correspond to the *engineUpdate*'s temporal logic. Then we construct FSAs to describe the relevant vulnerable program logic in PMS. Each edge in the FSA corresponds to a critical method invocation that brings the system state to the next state. For instance, Figure 5 is a simplified representation of one of the FSAs for the relevant vulnerable code in the PMS of Android version 4.4.4_r1. The completion of the *PMS.deletePackageLI()* invocation brings the state from STATE{"Replace Non-system Package Started"} to the STATE{"Old Package Deleted"}. Therefore, when we perform a model checking against the constructed CFG, whenever the system state STATE{"Replace Non-system Package Started"} encounters the *deletePackageLI()* method invocation in the PMS class file, our model checker will bring the current system state to the STATE{"Old Package Deleted"}. It then expects the next critical method invocation, namely *PMS.sndPkgBroadcast()*. Our model checker works in a depth-first-search manner to check the system states along the constructed CFGs. Whenever a sequence of method invocations is matched to a sequence of edges in the FSA that lead to the final STATE{"PMS Service Finished"}, a trace will be printed out for further verification. These traces help confirm the existence of the vulnerable logic for the AVD *engineUpdate* operation of the PMS component. We actually use the "liveness property" of model checking for vulnerability verification. Note that here we are to verify if something "good" will eventually happen, and here "good" means the identification of the exact vulnerable temporal logic in the PMS. Whenever the newly constructed CFGs of the PMS does not match the specified FSAs, we will check into the relevant CFGs to manually construct a new FSA for the unmatched PMS version. Most of the time, it is because the names of the methods are changed slightly, or a new wrapper is provided for an old method. It turns out that we only need to construct four FSAs to check the temporal logic of all PMS versions of Android OSes. The relevant mapping from FSAs to Android versions are given in Table 7. Due to the space limitation, we skipped some versions if our model checker reports no critical implementation variances.

Confirm the Goolge Android official factory Images.

To systematically check all Goolge Android factory images, we have downloaded all the factory images [9], which contain im-

Table 7: Android OS versions and relevant Models for PMS

OS versions	Release dates	Constructed FSAs
Android v4.4.4	19 June 2014	FSA of v4.4.4_r1
Android v4.4_r1	31 October 2013	FSA of v4.4.4_r1
Android v4.3_r2.1	14 July 2013	FSA ofv4.4.4_r1
Android v4.2_r1	13 November 2012	FSA of v4.4.4_r1
Android v4.1.2_r1	22 August 2012	FSA of v4.1.2_r1
Android v4.0.3_r1	16 December 2011	FSA of v4.1.2_r1
Android v2.3.7_r1	11 September 2011	FSA of v2.3.7_r1
Android v2.2_r1.1	29 June 2010	FSA of v2.3.7_r1
Android v2.1_r2	27 January 2010	FSA of v2.1_r2
Android v2.0_r1	14 October 2009	FSA of v2.1_r2
Android v1.6_r2	15 September 2009	FSA of v2.1_r2
Android v1.5_r4	30 April 2009	FSA of v2.1_r2

ages ranging from Android version v1.5_r4–v4.4.4_r1, totally 34 versions. Our model checker has verified that the same vulnerable logic in PMS appears from all the Android OS versions from v1.5_r4 (April 2009) to very recent v4.4.4_r1 (June 2014) for all Google official Android factory images.

Confirm the Vendor customized stock Images.

Due to the Android vendors' fragmentation problem [22, 5], we also download images from HTC and Samsung [17, 11] and conduct analysis on recently released stock images from multiple vendors, including Samsung (S4, S3 and S2 with Android v4.4.2, v4.0.4 and v2.3.4), HTC (One Mini, M7 with Android v4.4.2, v4.3), Sony (Xperia Z2 and Xperia M2 with Android v4.4 and v4.3) and LG (LGD958 Android v4.2.2). Our analysis also confirms that the same vulnerable PMS logic appears in all of them. It is because most of the customized stock images directly reuse most of the core design logic in mission critical components (e.g., PMS) from Google AOSP codebase.

Our finding demonstrates that some seemingly benign, but complicated updates in the system can cause surprising hazards to a stable system, which is similar to the takeaway message presented by Xing et al. [46]. But the newly discovered flaw resides in the PackageManager Service for AVDs' *engineUpdate* procedure, which can unexpectedly nullify any security protection from AVDs.

5. DISCUSSIONS

5.1 Mitigations

One proposed prevention for the discovered information leakage channels in Table 2 is based on more fine-grained mandatory access control. The Android system can leverage the existing SEAndroid [45] mechanism to restrict the public from accessing the channels under certain usage scenario with high security requirements. One specific policy refinement is to assign a new *subject domain* and relevant *allow* rules in the SEAndroid [45] policy database and then label a while-listed/authorized apps/services with the new subject domain. Consequently, only controlled/authorized accesses to different files under the $/proc/*$ (sub)folders are granted. However, this requires a solid understanding on different usage scenarios of various programs and the underlying Android system design specifics, so that the proposed mitigation will not block normal legitimate usability. One can model the relevant attacks in the mobile system into attack graph surface [33] and then enforce policy to block different leakage channels accordingly.

We propose another specific mitigation for the hazards in *heavy sweeping malScan*. For example, the underlying system may provide only the general usage statistics of AVDs (or other mission critical apps) by normalizing or delaying the reporting of system resource usage statistics (e.g., the CPU, memory and network) in relevant public channels in Table 2. Consequently, it is harder for an adversary to fingerprint an app's exact running status on the fly. The other way to reduce the risk is to introduce randomness into pre-scheduled scans. Instead of setting up an exact time, the user may specify a time window, in which an AVD can choose a random time to start scanning each day. A long-term mitigation is to improve the *malScan* operation and make scan operations *continuous* and *comprehensive* at the same time. According to our discussion with AVD vendors, this seems to be a future research topic, since *malScan* with both properties requires lots of testing and optimizations to be finally deployed on the resource limited mobile devices.

In addition to the designed short-term mitigations to AVD vendors in Section 4.2 for the *engineUpdate* hazard, we discuss with the Google security team that the system designers will have to rearchitect the PMS component to close or reduce the *null-protection window* in the app-updating procedure. One suggestion to Google is to delete the old APK only after the new process has been reactivated, but this might cause conflicts on app's package names or other app management tasks (e.g., the install verification process). Since lots of apps have continuous monitoring requirements (e.g., MDMs, IPSs, and etc.), Google has to find a way to resolve those conflicts for this feature enhancement in PMS.

5.2 Other Hazards

In our study, we find the other two potential hazards, which indicate that the discovered hazards are just a tip of the iceberg. So we discuss the other two types of hazards which can enable the evasions of current Android AVDs.

Cloud based malScan Hazard.

During the analysis, we sense a trend of adding the cloud-based scanning strategy for mobile platforms, including AVD # 1, 2, 3, 23 and 26. Cloud-based scanning fits resource limited mobile devices, as it can offload the heavy computation to a remote server by sending out the collected information. However, since the per UID network usage statistics can be directly accessed through the discovered channel # 16 in Table 2 as well, an adversary can plan evasions and attacks against AVDs using the similar fingerprinting strategy described in the *heavy sweeping malScan* hazard.

Virus definition file (VDF) update Hazard.

All the AVDs store their VDF and other files/data in the subdirectories, $/data/data/[AVD\_package]/*$, which are strictly set to be *world-unreadable* and enforced by the Linux kernel in Android. Our further analysis based on *Environment Information Collector* shows that this solid design of app data privacy protection is not enough for VDFs' deployment, as an adversary only needs to know the file size or other meta-data information of relevant files (e.g., created/updated time) in the subdirectory to infer the updating status of the VDF or other sensitive files (e.g., scan result caching file). We design a zero-permission app to call the *stat()* system call thorugh JNI to directly probe the meta-data information of all these files in an AVD's data folder. The whole path parameter that leads to different files (e.g., VDF) can be first collected via offline analysis. Knowing this status information, lots of potential targeted evasions can be designed. For instance, we find that some AVDs (e.g., # 21) perform a fresh *heavy sweeping malScan* right after the VDF is updated, so the adversary can drop or decrypt the newly obfuscated known malicious payloads a few minutes after the VDF has been updated, so as to make its fresh scan useless.

6. RELATED WORK

Malware analysis and threat prevention techniques [47, 38] have been designed and applied for offline analysis. Also, various interesting anti-analysis techniques have been discussed [42, 28] for both mobile and PC malware. Our proof-of-concept evasion techniques are conceptually similar to anti-analysis techniques, but we focus on a new angle to emphasize more on the evasion of AVD's online protection mechanism. Zhou et al. [53] provided a study of Android malware, and similarly, the discovered hazards in this paper are also based on a systematic study of 30 popular AVDs.

Android app and system hazards have been discovered in [41, 30, 46, 51, 50]. Accidental data disclosure between apps in mobile and PC systems have been discussed in [39]. The discovered information leakage channels for various hazards in AVD are relevant to the unexpected data exposure from the system side. Jana et al. [36] also take the per process memory usage and CPU scheduling statistics as probing channels to leak program's secrets. We identify that we can even leverage per thread usages to conduct fine-grained inference. Pileup attacks [46] are also based on a flaw in PMS that targets system update. Our *engineUpdate* hazard is based on one newly discovered flaw in PMS (related to the app-update mechanism). Empirical studies [30] have been performed on several hazards in security critical components or modules in Android apps. Our study is performed on current AVD apps, and several hazards have been discovered and reported to AVD vendors.

Antivirus evasion techniques [40, 24, 35] have been studied previously. Android Dalvik bytecode polymorphic transformation attacks have been presented by Rastogi et al. [44] to target incomplete signature database. However, our study concentrates on the the malware recognition mechanism itself, and the result shows that the quality of AVD's *malScan* mechanism should be further improved when deployed on the Android platform. Fedler et al. [31] discuss the lack of on-demand file system hooking problem of Android antivirus. Our study shows that some of the AVDs have already leveraged *FileObserver* APIs for that purpose. However, we find that it is the lack of combination of scan comprehensiveness and continuity that causes the ineffectiveness of the current design. What's more, the discovered hazards in the AVD *engineUpdate* are completely orthogonal to the file system hooking problem. Because any strong protections have to rely on an activated AVD process, which is missing in the *null-protection window*.

7. CONCLUSION

Based on an analysis framework, we conduct an empirical study of top 30 AVDs on the current Android platform. We discovered several serious hazards related to AVD malware scan mechanism, engine update procedure and etc. We then develop techniques to measure the feasibility of exploiting the hazards in *malScan* and confirmed the vulnerable *engineUpdate* program logic in the Android system through static analysis and model checking techniques. We reported the discovered vulnerabilities and hazards to AVD vendors, all of them have confirmed our findings and will take some of the mitigations suggestions in their latest versions. We also discuss the vulnerable design in the PMS that causing the *null-protection window* to the Google security team. They also admit the problem and will consider feature enhancement on the PMS component.

As the malware and the Android system keep evolving, more secure and preventive design strategies for mission critical apps (e.g., AVDs, IPS [12, 14], MDM [18] and etc.) should be adopted to reduce the chance of getting unexpected failures and loopholes.

8. ACKNOWLEDGMENTS

The authors want thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was partially supported by ARO W911NF-09-1-0525 (MURI), NSF CCF-1320605, NSFC 61100226 and NSF CNS-1223710. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and Army Research Office.

9. REFERENCES

[1] AndroGuard: Android Dalvik Bytecode Analysis Framework. http://www.blackhat.com/html/bh-ad-11/bh-ad-11-briefings.html.

[2] Android antivirus companies. Technical report. http://www.zdnet.com/android-antivirus-comparison-review-malware-symantec-mcafee-kaspersky-sophos-norton-7000019189/.

[3] Android Dalvik Debug Monitor Server. http://developer.android.com/sdk/installing/studio-tips.html.

[4] Android Monkeyrunner. http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html.

[5] Android OS Version Relative Chart ending on April 1, 2014 . http://developer.android.com/about/dashboards/index.html.

[6] AV TEST report, Jan 2014. http://www.av-test.org/en/tests/mobile-devices/android/jan-2014/.

[7] Avast! Mobile Security protects against USSD attacks. http://blog.avast.com/2012/10/04/avast-mobile-security-protects-against-ussd-attacks/ .

[8] DoS attack on Lookout mobile security application. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-3579.

[9] Factory Images for Nexus Devices. https://developers.google.com/android/nexus/images.

[10] Frequently updates of Antivirus Detection Engine, 2013. http://www.androiddrawer.com/15401/download-lookout-security-antivirus-8-30-1-app-apk/ .

[11] HTC Android Images from HTCdev. http://www.htcdev.com/devcenter/downloads/P00.

[12] Jinshan mobile duba. http://m.duba.net/.

[13] Kaspersky Lab Reports Mobile Malware in 2013. http://usa.kaspersky.com/about-us/press-center/press-releases/kaspersky-lab-reports-mobile-malware-2013-more-doubles-previous .

[14] LBE secrity guard. http://www.lbesec.com/.

[15] Prevent Unintended APP Update. http://www.symantec.com/connect/blogs/case-unintended-android-application-upgrade.

[16] Samli/Baksmali. http://code.google.com/p/smali/.

[17] Samsung Images from Samsung-updates. http://samsung-updates.com/.

[18] Samsung Mobile Device Management solution. http://www.samsung.com/global/business/mobile/solution/security/mobile-device-management/.

[19] The avast! AVD v2.0.4400 for Android allows attackers to cause a denial of service . http://cve.scap.org.cn/CVE-2013-0122.html.

[20] The Lookout AVD v8.17-8a39d3f for Android allows attackers to cause a denial of service . http://cve.scap.org.cn/CVE-2013-3579.html.

[21] The TrustGo AVD v1.3.6 for Android allows attackers to cause a DoS. http://cve.scap.org.cn/CVE-2013-3580.html.

[22] Android Platform Fragmentation. http://opensignal.com/reports/fragmentation-2013/, 2012.

[23] Frequently updates of Antivirus Detection Engine. http://m.aptoide.com/list/versions/com.lookout/83510, 2013.

[24] M. I. Al-Saleh and J. R. Crandall. Application-level reconnaissance: Timing channel attacks against antivirus software. In *4th USENIX Workshop on LEET 11'*.

[25] H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *NDSS*, 2004.

[26] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE*, pages 175–186, 2014.

[27] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*, 2013.

[28] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *DSN' 08*.

[29] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *S&P '05*.

[30] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS '13*.

[31] R. Fedler, M. Kulicke, and J. Schutte. An antivirus API for Android malware recognition. In *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013*.

[32] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *MobiSys '13*.

[33] H. Huang, S. Zhang, X. Ou, A. Prakash, and K. Sakallah. Distilling critical attack graph surface iteratively through minimum-cost sat solving. In *Proceedings of the 27th ACSAC*, pages 31–40. ACM, 2011.

[34] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*. Springer, 2013.

[35] S. Jana and V. Shmatikov. Abusing file processing in malware detectors for fun and profit. In *SP' 12*.

[36] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *S&P '12*.

[37] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *CCS '14*.

[38] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, 2009.

[39] A. Nadkarni and W. Enck. Preventing accidental data disclosure in modern operating systems. In *CCS '13*.

[40] J. Oberheide, M. Bailey, and F. Jahanian. PolyPack: an automated online packing service for optimal antivirus evasion. In *3rd USENIX on Offensive technologies*.

[41] J. Oberheide and F. Jahanian. Remote fingerprinting and exploitation of mail server antivirus engines, 2009.

[42] G. Pék, B. Bencsáth, and L. Buttyán. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11.

[43] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS '14*.

[44] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *asiaCCS*. ACM, 2013.

[45] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, 2013.

[46] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *IEEE Symposium on S&P 14*.

[47] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Sec '12*.

[48] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of ACM WiSec '14*.

[49] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of CODASPY '12*. ACM.

[50] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: inferring your secrets from android public resources. In *In ACM CCS*. ACM, 2013.

[51] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *IEEE Symposium on S&P*, 2014.

[52] Y. Zhou and X. Jiang. An analysis of the anserverbot trojan. http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBotAnalysis.pdf.

[53] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *SP '12*. IEEE.