The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations

Xiaoyong Zhou*, Yeonjoon Lee*, Nan Zhang*, Muhammad Naveed† and XiaoFeng Wang*
*School of Informatics and Computing
Indiana University, Bloomington
Email: {zhou, yl52, nz3, xw7}@indiana.edu
†Department of Computer Science
University of Illinois at Urbana-Champaign
Email: naveed2@illinois.edu

***Abstract*—Android phone manufacturers are under the perpetual pressure to move quickly on their new models, continuously customizing Android to fit their hardware. However, the security implications of this practice are less known, particularly when it comes to the changes made to Android's Linux device drivers, e.g., those for camera, GPS, NFC etc. In this paper, we report the first study aimed at a better understanding of the security risks in this customization process. Our study is based on ADDICTED, a new tool we built for automatically detecting some types of flaws in customized driver protection. Specifically, on a customized phone, ADDICTED performs dynamic analysis to correlate the operations on a security-sensitive device to its related Linux files, and then determines whether those files are under-protected on the Linux layer by comparing them with their counterparts on an official Android OS. In this way, we can detect a set of likely security flaws on the phone. Using the tool, we analyzed three popular phones from Samsung, identified their likely flaws and built end-to-end attacks that allow an unprivileged app to take pictures and screenshots, and even log the keys the user enters through touchscreen. Some of those flaws are found to exist on over a hundred phone models and affect millions of users. We reported the flaws and helped the manufacturers fix those problems. We further studied the security settings of device files on 2423 factory images from major phone manufacturers, discovered over 1,000 vulnerable images and also gained insights about how they are distributed across different Android versions, carriers and countries.**

I. Introduction

The Linux-based open source Android platform has grown into the mainstay of mobile computing, attracting most phone manufacturers, carriers as well as millions of developers to build their services and applications (*app* for short) upon it. Up to August 2013, Android has dominated global smartphone shipments with nearly 80% market share [8]. Such success, however, does not come without any cost. The openness of the system allows the manufacturers and carriers to alter it at will, making arbitrary customizations to fit the OS to their hardware and distinguish their services from what their competitors offer. Further complicating this situation is the fast pace with which the Android Open Source Project (AOSP) upgrades its OS versions. Since 2009, 19 official Android versions have been released. Most of them have been heavily customized, which results in tens of thousands of customized Android branches coexisting on billions of mobile phones around the world. This fragmented ecosystem not only makes development and testing of new apps across different phones a challenge, but it also brings in a plethora of security risks when vendors and carriers enrich the system's functionalities without fully understanding the security implications of the changes they make.

Security risks in customizations. For each new Android version, Google first releases it to mobile phone vendors, allowing them to add their apps, device drivers and other new features to their corresponding Android branches. Such customizations, if not carefully done, could bring in implementation errors, including those with serious security consequences. Indeed, recent studies show that many pre-loaded apps on those images are vulnerable, leaking system capabilities or sensitive user information to unauthorized parties [43]. The security risks here, however, go much deeper than those on the app layer, as what have been customized by vendors are way beyond apps. Particularly, they almost always need to modify a few device drivers (e.g., for camera, audio, etc.) and related system settings to support their hardware. In our research, we found that most customizations on the Android kernel layer are actually related to those devices (Section II-B), and they are extremely error-prone, due to the complexity of Android architecture and the security mechanism built upon it.

Android is a layered system, with its app layer and framework layer built with Java sitting on top of a set of C libraries and the Linux kernel. Device drivers work on the Linux layer and communicate with Android users through framework services such as Location Service and Media Service. Therefore, any customization on an Android device needs to make sure that it remains well protected at both the Linux and framework layers, a task that can be hard to accomplish within the small time window the vendors have to develop their own OS version. Any lapses in safeguarding these devices can have devastating consequences, giving a malicious app access to sensitive user information (e.g., photos, audio, location, etc.) and critical services they provide (e.g., GPS navigation). However, with the complexity of Android's layered system architecture and limited device-related documentations available in the wild, so far, little has been done to understand the security risks in such device customizations, not to mention any effort that helps detect the threats they may pose.

Flaw detection. In this paper, we report the first systematic study on the security hazards in Android device customizations. Before we go to the details of this research, a few terms used throughout the paper are explained in Table I. Our

TABLE I. TERMINOLOGIES IN THIS PAPER

Term	Semantic
Phone	Mobile phone, e.g., Nexus 4, Samsung Galaxy SII. Here we avoid using "device" to refer to phone.
Device	Hardware on the phone, e.g., camera, GPS.
Device node or device file	An interface for a device driver, e.g., `/dev/null`.
Device-related file	Files related to device operations, including device nodes and other files such as logs.

study is based upon an automatic tool, *ADDICTED*, which we designed to detect such customization hazards. The high-level idea is to automatically identify the Linux files related to the operations on the devices (e.g., phone's camera) Android intends to protect, and compare the levels of protection (in terms of Linux file permissions) for the individual files on a vendor's version with those on the corresponding AOSP version (called a *reference* in our research[1]). The rationale here is that a Linux file related to a security-critical device operation on the customized Android should not be less protected than its counterpart on the reference. Otherwise, it can lead to a security hazard. For example, we do not expect that the device node (see Table I) for camera on the Samsung Galaxy SII is set publicly readable and writable (a Linux permission of `666`), while its counterpart on Android 4.0.4 is accessible only by `system:camera` user group (`660`). Actually, even if we do not know the semantic of the file (i.e., the camera node), the presence of the discrepancy in its Linux permission settings across the two OSes, together with its relation with a dangerous Android permission on devices (camera access), is sufficiently alarming to justify a close attention.

To implement this idea, we built into ADDICTED a component called *Device Miner* that dynamically maps permission-protected device operations (through APIs such as `Camera.startPreview()`) on the Android framework layer to their related files on the Linux layer. This requires monitoring the way Android processes device-related service requests across multiple layers (framework, libraries, Linux and even hardware) over different phones, in the presence of complicated event and call-back mechanisms, which cannot be handled by existing techniques [26, 44]. Our design addresses these technical challenges with a simple differential analysis over the system call traces (recorded by strace [16]) with and without a specific device-related operation. Device Miner can further fingerprint a device node with a set of system calls involving the file and their parameters. The fingerprint serves to correlate device-related files on a customized phone to those on its reference (the Nexus reference), even when they have different names and are under different Linux groups. Over such correlation information identified by Device Miner, another component, *Risk Identifier*, detects *likely customization flaws* (LCFs, i.e., a downgrade of the Linux protection level of a customized device-related file) through the aforementioned comparison (between the customized phone and its Nexus reference).

We evaluated ADDICTED on a Google Nexus 4 with 4.2 and 4.3 and Samsung customized Android on Galaxy SII, ACE 3 and GRAND. Our analysis discovered 4 LCFs across those phones: oftentimes, device nodes that are supposed to be protected to the system level have been exposed to the public on the Linux layer; examples include `input`, `camera` and `frame buffer`. We further conducted a case study on three discovered LCFs and constructed end-to-end attacks in which we take pictures and screenshots, and read touch-screen coordinates on the phones without requesting relevant Android permissions. As a prominent example, our analysis found that on Samsung Galaxy SII, camera device node has been made publicly accessible. To demonstrate the devastating consequences once this vulnerability is exploited, we built a carefully drafted attack app that directly commands the camera driver to take pictures without being noticed by the phone user, even when it does not have the permission to do so. This problem affects over 40 million Galaxy SII users alone [12]. We reported our findings to Samsung, who acknowledged the importance of this research and our findings. We are now working with them to fix those problems. Video demos of our attacks can be found online [1].

Large-scale measurement. To understand the scope and magnitude of such security hazards in device customizations, we further scanned over 2423 factory images from Samsung, LG and HTC for LCFs, through searching for the device-related files on those images using the file names from AOSP images, and inspecting the levels of their Linux-layer permission protection. This large-scale measurement study led to the discovery of 1290 problematic images and hundreds of under-protected devices. It also yields several new interesting findings not known before. For example, we found that most device-related LCFs on Samsung Galaxy SII have never been fixed, even though its operating system was upgraded from 4.0, to 4.1 and later to 4.2. As another example, our research shows that the customizations to the smartphones distributed in China and Brazil have more potential security flaws than those sold in North America. We also measured the LCFs discovered on different carriers' customized phones and identified the most vulnerable ones.

Contributions. We summarize the contributions of the paper as follows:

- *New techniques*. We developed a new technique that made the first step towards automated discovery of security-critical vulnerabilities introduced during Android device customization. The technique leverages system-call level information to link the device operations protected by Android permissions to their related Linux files, and a simple differential analysis to detect the potential customization flaws in those files' protection. Our approach helps discovery of previously-unknown critical security flaws in real-world mobile systems, which affect millions of smartphone users.

- *New findings*. We performed the first large-scale measurement study on the security implications of Android device customizations. Our research reveals a large number of potential security flaws within hundreds of customized images, and sheds light on a few important issues such as how such security risks are distributed across different countries, carriers and Android versions.

[1]In our research, we actually used Google's customization of AOSP (for Nexus phones) as the reference, as Google's version closely follows the original AOSP OS with minimum changes being made.

Roadmap. The rest of the paper is organized as follows: Section II introduces the background of Android vendor customization and a study we performed to understand the layers of Android most heavily customized; Section III presents the design and implementation of ADDICTED; Section IV describes a case study in which we analyzed customized devices using ADDICTED and discovered high-impact security flaws; Section V reports our large-scale measurement study on the security flaws in device customizations; Section VI discusses the limitation of the work and potential future research; Section VII reviews the related research and Section VIII concludes the paper.

II. Vendor Customizations on Android

In this section, we provide some backgrounds for our research and elaborate a study that helps to understand the way Android is customized by different vendors.

A. Background

Android architecture and security model. As discussed before, Android has a hierarchical architecture. On top of the stack are various Android apps, including those from the system (e.g., contacts, phone, browser, etc.) and those provided by third parties. Supporting these apps are the services running on the *framework* layer, such as Activity Manager, Content Providers, Package Manager, Telephone Manager and others. Those services mediate individual apps' interactions with the system and enforce security policies when necessary. The nuts and bolts for them come from Android `C` libraries, e.g., SSL, Bionic, Webkit, etc. Underneath this layer is the Linux kernel, which is ultimately responsible for security protection.

The Android security model is built upon Linux user and process protection. Each app is given a unique user ID (UID) and by default, only allowed to touch the resources within its own sandbox. Access to system resources requires permissions, which an app can ask for at the time of installation. Decisions on granting those permissions are made either by the system through checking the app's signatures or by the user. When some permissions are given to an app, it is assigned to a Linux group corresponding to the permission such as `gps`. Resources on Android typically need to be protected on both the framework layer and the Linux layer: the former checks an app's permissions and the latter is expected to enforce security policies consistent with those on the framework layer to mediate the access to the resources.

Vendor customization. Android is an open system. Google releases the AOSP versions as baselines and different manufacturers (e.g, Samsung, HTC etc.) and carriers (e.g., AT&T, Verizon etc.) are free to tailor it to their hardware and add new apps and functionalities. Most of these Android versions from the vendors have been heavily customized. For example, prior research [43] shows that among all the apps pre-installed by the major smartphone vendors (Samsung, HTC, LG, Sony) on their phones, only about 18% come from AOSP, and the rest are either provided by the vendors (about 65%) or grabbed from third parties (17%). Under the current business model, those vendors have a small time-window of about 6 months to customize the official version. This brings in a lot of security issues: it has been reported that over 60% of the app vulnerabilities found in a study come from vendor customizations [43].

The primary reason for vendors to customize Android is to make it work on their hardware. Therefore, the most heavy-lifting part of their customization venture is always fitting new device drivers to the AOSP baseline. This is a delicate operation from the security viewpoint: not only should those new drivers be well connected to their corresponding framework layer services, so that they can serve apps and are still protected by permissions, but they also need to be properly guarded on the Linux layer. Further complicating the situation is the observation that a new device may require its driver to talk to other existing drivers. The problem here is that the latter's permission settings on AOSP could block such communication. When this happens, the vendor has to change the driver's security settings on Linux to accommodate the new driver. An example is the camera device on Galaxy SII that needs to use the `UMP` driver to allocate memory; for this purpose, Samsung made `UMP` publicly accessible. So far, the security implications of this customization are unclear.

Android Linux devices. Android inherits the way Linux manages its device drivers and related files, in which both block devices (e.g., flash drives) and stream devices (e.g., virtual terminals) are placed under `/dev`. Other devices like network devices are placed under other directories (e.g., `/sys`). In addition to those standard devices, Android further introduces a plethora of new devices, such as camera, accelerometer, GPS, etc., whose proper levels of security protection on the Linux layer have never been made public. These new devices have been heavily customized by vendors, who also bring in an array of perplexing device-related files for their new hardware pieces like CPU, graphic device, etc. Some of such devices are security-critical, which are protected under Android permissions or not even made available to apps such as `/dev/graphic/fb0`.

Those devices, if inadequately protected, can allow an unauthorized app to get access to sensitive user data (e.g., locations, conversations, etc.) or critical system capabilities (e.g., taking pictures). However, it is challenging to find out whether they are guarded to the level expected, as simply correlating devices to their device nodes under `/dev` (or `/sys`) can be hard. Except a few standard Linux devices, Android never provides any documentation to explain the device-related files under those directories. Different vendors further add their customized device nodes, arbitrarily change their names, group affiliations, etc. As a result, even finding the counterpart for a given AOSP device node on a customized OS can be difficult. As an example, the near field communication (NFC) device node is `/dev/bcm2097x-12c` on the Nexus 4 Android 4.2, while on Samsung SII, it becomes `/dev/pn544` as detected by ADDICTED. Without knowing the relations between Android Linux devices and their drivers, little can be done to find out whether they are properly protected.

In our research, we made the first, though preliminary, step toward a better understanding of the security risk in customizing Android Linux devices. Our idea is to use a dynamic analysis to map a set of security-critical devices (in Table IV) to their related Linux files, and then compare the protection they receive with that provided on an AOSP Android

(again, Google-customized OSes in our research). This helps us assess whether certain important devices become under-protected during a customization (Section III).

Adversary model. The purpose of our research is to understand the security risks in customizing Android Linux devices. To evaluate such risks, we assume the presence of malicious apps on the phone running a customized Android OS. These apps do not have root privileges, nor do they have the permissions to use the devices under investigation, such as camera, audio, GPS, etc. On the other hand, they are actively seeking access to those protected devices through exploiting the vulnerabilities introduced by the customization. We want to understand whether such attempts can be successful.

B. Understanding Customizations

During a customization, the vendor could change any Android layer. In our research, we performed a study to understand which layer has been heavily modified. The study shows that changes mainly happen on the app layer and Linux layer and rarely does the vendor touch the Android framework interface such as services. On the Linux layer, most effort has been made on device drivers to support new hardware. Here we elaborate this study.

Methodology. To find out where modifications happen, we compared the source code of two popular vendor-customized phones – Samsung Galaxy SII (AT&T version i.e. SGH-I777) and Samsung Galaxy Ace 3 (GT-S7270L) – with their corresponding AOSP versions. Specifically, we paired the source code `GT-S7270_JB_Opensource` for Samsung Galaxy Ace 3 with its AOSP reference Android 4.2 and kernel `android-msm-mako-3.4-jb-mr2`, and `SGH-I777_NA_JB_ATT_Opensource` for Samsung Galaxy SII with its AOSP reference Android 4.0.4 and kernel `android-samsung-3.0-jb-mr0`. For each pair, we used a diff-tool (DeltaWalker [3]) to measure how many files have been added/modified/deleted under different directories during Samsung's customization of the AOSP code. DeltaWalker is a file and folder comparison and synchronization tool. In our study, it was configured to compare files based upon text (line by line), as opposed to individual bytes. Before our analysis, we also filtered out files related to version control (e.g., `.git` or `.gitignore`), and tuned the tool for accuracy, instead of speed. The outputs of the analysis were sanitized to remove the differences caused by whitespace and delimiters.

Findings. The results of the study are presented in Table II and Table III. As is clear from the tables, changes made on the Linux layer mainly happen to the driver source-code directory, which involve hundreds or even thousands of files being added, modified or deleted. The other directory with the similar dynamic is `arch`, which contains different hardware-related source code for processors. Actually, other directories also contain code related to device drivers. However, even just looking at the `driver` directory, we found that its modifications are extensive compared with other directories. On the framework layer, most of the customizations are either related to device (`/device/samsung/bcm_common`) or new apps (`/vendor/samsung/common/packages`), as presented in Table III, while the service files under `/framework` have not been touched at all.

TABLE II. CUSTOMIZATIONS IN LINUX KERNEL

Path	Galaxy Ace 3 added/modified/deleted	Galaxy SII added/modified/deleted
/arch	1202/341/1029	695/244/407
/block	0/10/1	0/7/0
/crypto	0/0/0	1/12/0
/drivers	**958/661/1390**	**2830/687/322**
/firmware	1/1/0	45/1/1
/fs	0/74/ 0	0/34/0
/include	210/183/213	135/306/14
/init	0/2/0	0/2/0
/kernel	0/44/3	0/25/1
/lib	0/12/1	0/6/0
/mm	0/22/0	10/23/0
/net	0/84/20	108/26/0
/scripts	3/5/2	3/1/0
/security	0/5/0	0/6/0
/sound	339/29/99	68/31/15
/tools	0/12/1	0/0/0
/virt	0/1/0	0/0/0

TABLE III. CUSTOMIZATIONS IN ANDROID FRAMEWORK LAYER

Path	Galaxy Ace 3 added/modified/deleted	Galaxy SII added/modified/deleted
/device/samsung/bcm_common	**2395/0/0**	
/external/bluetooth		3/26/0
/external/chromium	2/38/1	0/0/19
/external/dnsmasq	35/44/0	0/4/0
/external/e2fsprogs	160/436/24	1/2/2
/external/iproute2	0/0/0	1/0/0
/external/iptables	3/0/3	0/0/28
/external/KeyUtils		3/0/0
/external/libexifa		60/0/0
/external/libjpega		40/0/0
/external/webkit	69/525/0	31/238/0
/libcore	0/3/0	
/packages/apps/BluetoothTest		2/0/0
/packages/apps/.../mozilla		58/0/0
/packages/apps/Email/lib_Src		807/0/0
/vendor/broadcom/common	3/0/0	
/vendor/samsung/.../external	4/0/0	
/vendor/samsung/.../frameworks	102/0/0	
/vendor/samsung/.../packages	**35936/0/0**	
/framework	**0/0/0**	**0/0/0**

III. AUTOMATED DETECTION OF SECURITY FLAWS IN DEVICE CUSTOMIZATIONS

Our study shows that besides pre-installed apps, Linux device drivers are the focus of vendor customizations (Section II-B). To better understand the security risks that come with such customizations, we designed and implemented ADDICTED (Android Device Customization Error Detector), a suite of new techniques for automatic detection of the problems in customized device protection. In this section, we first describe our high-level idea and then present the details of our techniques.

A. Overview

The design. Given a set of security-critical Android devices, it is nontrivial to find out whether they are well-protected on the Linux layer. Although, we may figure out the rough locations of their related Linux files (most likely under `/dev`), finding them in hundreds of files is difficult. Even more complicated is the evaluation of their protection levels under Linux, which needs semantic information about what those files indeed are (device nodes, log files, etc.). To address these issues, towards automatic detection of customization flaws, we propose running a dynamic analysis to identify all the files

related to a customized device and then comparing the Linux permission settings of those files with their counterparts on the AOSP reference (the Android version installed on Google Nexus phones). The rationale here is that there is no reason for the vendor to lower the protection levels of such files, particularly when they are related to a security-critical device. The risks discovered in this way (i.e., LCFs) then need to be further investigated to understand their security implications.

Based upon this idea, our design receives from the user a list of devices to be analyzed. The Linux files for some of those devices could be well-known, such as the input device (`/dev/input/event*`) and the frame buffer (`/dev/graphics/fb*`) [7, 11], but most of them are Android additions and therefore, less known (e.g., drivers for camera, NFC, etc.). To identify these files, ADDICTED runs a suite of test cases that serve as an input to the dynamic analyzer. The analyzer traces the execution of the Android system when it is processing these cases and operating on their related devices, in an attempt to catch all the files necessary for such operations. Each of these files is further fingerprinted by the way they are handled (e.g., system call types and parameter values). The outcomes of this analysis on an AOSP version (including files related to different devices and their fingerprints) serve as a reference. Such reference files are correlated to those discovered from a customized Android based on their fingerprints, and further compared with them in terms of individual files' Linux permission settings. Once any discrepancy is found, particularly when the customized file has a lower protection level (e.g., system-only for the file on AOSP and publicly readable on the customized version), a security risk is reported.

Architecture. Figure 1 illustrates the design discussed above. ADDICTED includes *Device Miner*, that performs the aforementioned dynamic analysis on the test cases running on a customized Android phone, and further analyzes its outputs (including the system-call traces from multiple executions of individual cases) to identify a set of files related to each device. Those files and their fingerprints are then handed over to *Risk Identifier*, which correlates them to those on the reference AOSP version and reports an LCF once any of them is found to be under-protected.

B. Device Miner

The approach. Device Miner is designed to trace operations on an Android device to identify its related Linux device files. Given the complexity of the Android architecture, this is by no means trivial. Specifically, static analysis of Android is complicated, given its layered structure with the framework written in Java and the Linux kernel in C. When it comes to dynamic analysis, existing tools like TaintDroid [26] can support a variable and message level taint analysis. However, to handle the complicated inter-process communication (IPC) and message passing model within Android, it requires intensive instrumentations of the OS. As a result, it becomes less portable and unsuitable for analyzing a large number of customized phones. Alternatively, DroidScope [44] performs the analysis within a virtual machine, however, it cannot conveniently simulate different types of hardware on customized phones or tablets.

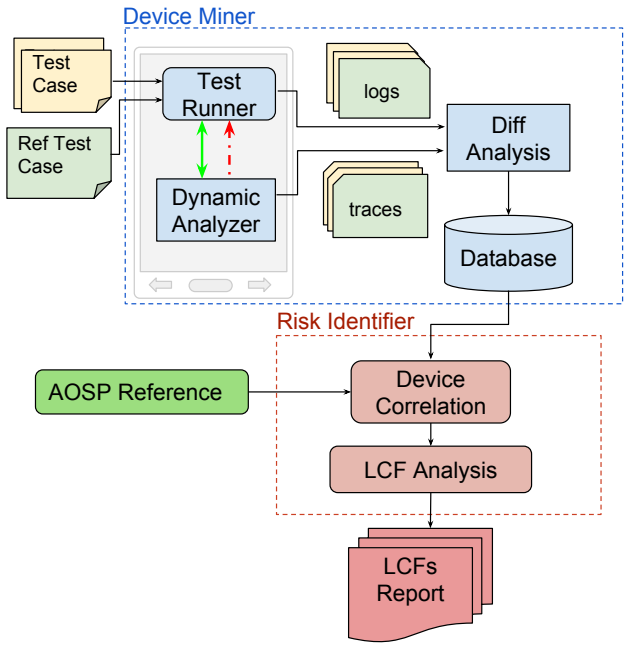

Fig. 1. Design of ADDICTED. Test cases are first executed by Test Runner and analyzed by the Dynamic Analyzer. The logs and traces are analyzed by the Diff Analysis tool to filter out irrelevant device operations and later sent to Risk Identifier to check for LCFs.

In our research, we built into Device Miner a dynamic analyzer that works on the system-call level. This makes it coarse-grained in tracking device operations, but much more lightweight and portable than prior approaches. More specifically, Device Miner utilizes a suite of test cases to trigger device operations such as taking pictures, requesting geolocations, etc., and attaches *strace* [16] to the app that runs those cases. The app does not directly access device files sitting on the Linux layer, and hence it needs to send an IPC to acquire relevant OS services to access them as shown in Figure 2. To find out the service that operates on a given device, our approach leverages an instrumented *binder* to follow the IPC call and identify the system process that serves the app's request, and then attaches strace to that process and its children. On a customized device, this step can be replaced by directly attaching straces to the processes and services involved in execution of a test case, based upon a model identified from running the same case on a reference phone, so as to avoid modifying any OS code on the device. In this way, Device Miner is able to observe all system level activities when Android is operating on a target device, and find out all the files it touches. We further perform a differential analysis to remove those unrelated to the device operations. In Figure 2, we elaborate individual components of Device Miner.

Test cases. As an input to ADDICTED, we need to prepare a list of security-critical devices. The devices we tested are shown in Table IV. This list covers most (if not all) of the common Android hardware protected by Android permissions[2] (at the `dangerous` level). Also on it are some standard Linux devices, whose device nodes are well known. Those devices

[2]Note that we did not include nonsensitive Android devices such as gyro, accelerometer.

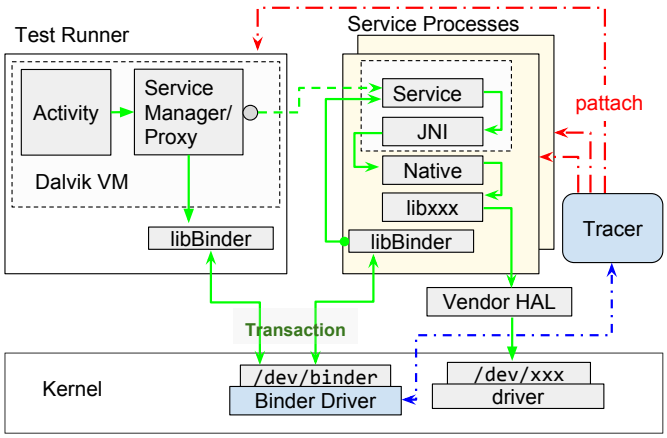

Fig. 2. Dynamic Analyzer. In the figure, the dashed green (---) line describes a conceptual RPC call and the solid green line (—) is the real call chain for accessing hardware devices. The dash-dash-dot red (– · – · –) lines show how the tracer is attached to all the processes related to a device operation and the dash-dot blue (-.-.-) line illustrates how this attachment operation is performed by the binder.

TABLE IV. SENSITIVE DEVICES: ANDROID-SPECIFIC OR LINUX-INHERITED

Device	Related Permissions	Test Operations or Device Node
Camera	CAMERA	Take pictures, change camera settings
NFC	NFC	Send NFC tags
Audio	RECORD_AUDIO	Play audio, record audio
Radio	CALL_PRIVILEGED CALL_PHONE	Make phone calls
External Storage	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE	Write and read from external storages
GPS	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION	Read GPS data
Bluetooth	BLUETOOTH BLUETOOTH_ADMIN	Pair to a device, send a file
Wifi	ACCESS_WIFI_STATE	Enable wifi, connect to wifi
Frame Buffer		/dev/graphics/fb*
Input Subsystem		/dev/input/event* /dev/uevent
Block Devices		/dev/block/*
VPN, PPP	BIND_VPN_SERVICE	/dev/vpn, /dev/ppp

are not analyzed by Device Miner and instead, directly sent to Risk Identifier for security checks. Note that even though this list is not complete, the same methodology can be used to evaluate other devices, once their test cases are added, to improve the coverage of the analysis.

For each device on the list (except those standard Linux devices), we built a test case, which is executed by Test Runner. The app calls related APIs to operate on the device, such as taking a picture, transferring a tag for an NFC device, etc. Such a test case can be generated automatically using the testing tools such as Randoop [13]. However, given the relatively small number of devices and the complexity in automatic construction of a correct call sequence and parameters, we just manually developed those cases for our implementations. Once this test suite is constructed, it can be executed upon different customized Android phones automatically.

Dynamic analysis. As soon as Device Miner starts the Test Runner, it attaches a *tracer* (a wrapper of strace) to the app's process. The app needs to make API calls to access its target devices. In Android, such an API call goes through the binder driver in the kernel, which passes the request to a system service. This interaction is called a *transaction* as shown in Figure 2. Device Miner includes an instrumented binder that monitors the processes communicating with the test app and attaches tracers to them.

Specifically, our approach instruments `binder.c` with the code for inspecting individual transactions. This can be done automatically, given the binder's source code has not been changed significantly across different Android versions. Later we will discuss an alternative to avoid even this mostly automated instrumentation.

During its runtime, this modified binder checks the transaction parameters of an IPC call to extract the source Process Identifier (PID) of the transaction and its target PID, together with the transaction data. The source PID is directly retrieved from `binder_proc.pid` and the target one is obtained from `target_node`, which is referred by `target_handler` embedded in the transaction data. Those PIDs are further mapped to package names retrieved from process memory using `access_process_vm`. If either party in the IPC (source or target) is found to be the test app, based upon its package name, Device Miner attaches a tracer to the one that has not been monitored yet. Those tracers log all their processes' file operations, such as `ioctl`, `read` and `write`. They are also capable of parsing some parameters for the file operations, which are important to fingerprinting a file (Section III-C).

A problem for our dynamic analysis is instrumentation of the binder. Although this can be done automatically on Android source code, we still need to compile the instrumented code and install it on every customized device, which makes the approach less portable. To address this issue, we further leverage the way the AOSP OS handles a service to simplify the analysis. Specifically, as discovered in our research (Section II-B), mobile phone vendors rarely modify the framework layer services during their customizations. Particularly, they tend to leave the package names of the services intact, because otherwise, a large number of program locations referring those packages need to be adjusted as well. Therefore, we can assume that the names of the services working on a device-related request stay unchanged across different Android OSes (though the names of device drivers could be different). In this case, Device Miner can build a model for running each test case, based upon what it observes on the AOSP OS (the reference), to record all the packages involved in handling its device-related requests. On a customized system, our approach automatically attaches tracers to all the processes of those packages during the operations of the test case to monitor files they drop. Since we do not need to change the OS to run strace, the whole Device Miner becomes completely portable across different Android phones. There could be a problem when some vendors indeed touch the framework layer. In this case, we need to install the instrumented binder on the device and run the test again.

Differential analysis. To achieve portability, Device Miner tracks device operations at the system-call level, which is coarse-grained. For the Linux files discovered this way, we

do not know whether they are indeed related to the device or just the noise introduced by other processes running in the background. To remove such random noise, our approach further performs a *differential analysis* on the outcomes of an analysis, comparing them with those produced by other independent tests. Specifically, we run the same test case multiple times, independently, until the set of the files touched by all those executions no longer change (which indicates that those files are related to the test case, not other processes). This filters out files that are touched by other concurrently running processes. Then, we invoke the test case again, and this time, the app attempts to access the target device *without* a proper permission and naturally will not get what it asks for. The outputs we observe from this execution are used to remove the files from the intersection produced by the prior runs, since those files do not contribute to the normal operations on the target device. In this way, we get a "sanitized" list of files that are very likely to be directly related to the device.

C. Risk Identifier

Given a list of Linux files related to a device, as identified by Device Miner, we need to find out whether their security protection levels are properly set. The problem is that even though we know the device protected by a dangerous or signature permission on the framework layer, we still have no idea whether the Linux permissions assigned to the files are appropriate, given that there is little semantic information about what those files indeed are. For example, it would be natural to have `null` device node less protected than other device nodes. Even given such semantic information, AOSP has never made it clear how to set the Linux permissions for a device-related file properly based on its protection at the framework layer. To work around such complexities, in our research, we simply take the way those device-related files are configured on AOSP OSes as references and compare the security settings of a file on a customized OS with its counterpart in the reference. Note that we do not assume here that AOSP always makes device protection right on the Linux layer, though the references are typically less error-prone than their customized counterparts. Our point here is that if a customization makes a highly-protected file related to a security-critical device on the original AOSP version less protected, there could be a security issue (i.e., a LCF). Also, nor do we consider that our approach can find us most customization errors: after all, it cannot handle new devices and their files that never show up on the reference OS. Nevertheless, our study demonstrates that even this simple, first-step approach can already detect security-critical and also high-impact vulnerabilities in customizations (Section IV).

To implement this idea, we need to correlate device files on different phones and detect LCFs through the aforementioned differential analysis. Risk Identifier is designed for this purpose. Here we elaborate how it works.

Device file correlations. To customize an AOSP version to fit their hardware, vendors typically need to add to the system their own device drivers and other supporting files. Compared with their AOSP counterparts, these files could have different names and be assigned to different Linux groups, as observed in our study (Section IV). For example, the camera device node on Nexus 4 (with an Android 4.2) is `video0` or `video1` under the group `camera`, while on GRAND, it becomes `vc-cam` and affiliated with the Linux group `system`. Correlating such a file to their counterpart on a reference OS can become nontrivial when Device Miner outputs multiple device-related files on both sides. Here we describe how Risk Identifier establishes such a connection.

Our approach first fingerprints those files based upon how they are operated by system calls. Specifically, for each file, we look at the set of system calls that touch the file, and the content and other features of those calls' arguments. For the arguments that pass values to a system function, strace can often parse them to find their content. Since the designs of customized devices (camera, video, Bluetooth, etc.) tend to follow their industry standards, oftentimes, the content of the arguments for their related system calls can be informative enough for establishing the relation between two device nodes built according to the standards. For example, different camera devices designed according to the V4L2 driver framework [17] all share some arguments for the `ioctl` call that operates on them, such as `0x560f` (parsed into `VIDIOC_QBUF`). Therefore, whenever two device-related files are found to have some of such standard arguments (automatically identified by strace) in common, they are considered to be related. In the case that Risk Identifier cannot find such common arguments, it further checks the set of system calls that happen to each file to connect customized file to the reference one. This approach works particularly well on the device whose operations need to go through a standard procedure. An example is NFC, whose call sequence is always (`select, read, write`) with the 2nd arguments of `read` and `write` being well-formatted NFC streams, even when the drivers are heavily customized, with different file names and group memberships. Note that given a device node on the reference side, all we need to do here is just to determine which customized file (out of several ones reported by Device Miner) is more likely to be its counterpart. This can often be done, since Android processes device nodes differently from other files, such as logs.

LCF detection. After pairing individual device files on a customized phone with those in a reference, an analysis tool starts checking their Linux file permissions. For a pair of device files, what we are looking for there is any discrepancy in their permission settings. For example, if the one within the reference is made readable and writable only to group members while the other is open to the public, then our approach immediately reports discovery of an LCF. More complicated is when these two files end up in different Linux groups: for example, one in `camera` and the other in `system`. We consider this practice *risky*, though it may not lead to any exploitable flaw. The rationale is that Android maps permissions to different Linux groups and as a result, change of a resource's group affiliation could open an unexpected avenue for unauthorized access. Of course, in most cases, such affiliation changes may not result in security-critical vulnerabilities. A device file found to have this problem is thus just marked as *risky*, and will not be alarmed as an LCF.

IV. Findings and Attacks

In our study, we ran our implementation of ADDICTED on four smartphones to analyze the security protection of their Linux devices. These phones include a Google Nexus 4 with

an Android 4.2, a Samsung Galaxy SII with a customized 4.0.3, a Galaxy ACE 3 with a 4.2.2 and a Galaxy GRAND with a 4.1.2. ADDICTED instrumented the binder on the Nexus 4 to identify the device files for our test suites. It also built up a model for each test case, recording the packages involved in handling its device access request. All the device files discovered from the rest three phones were compared with their counterparts on the Nexus 4, which served as the reference, to identify their LCFs based on their Linux permission settings.

This study discovered 4 LCFs and all of them were confirmed to be indeed problematic. Particularly, we performed an in-depth study on 3 such flaws[3] and came up with end-to-end attacks on them. These attacks enable a malicious app, without relevant permissions, to log the keys the phone user enters on her touchscreen, and to take pictures or screenshots stealthily. Some of these flaws were found to be extremely pervasive, affecting millions of smartphone users, as discovered by our measurement study reported in Section V. Following we elaborate this research.

A. Findings

Device files identified. On the Nexus 4 (the reference), ADDICTED automatically identified the files associated with individual devices. Table V illustrates our findings. For `NFC` devices, only a single file was found for each of them, which turned out to be their device nodes. Other devices are more complicated. For example, on Nexus 4, the front video device is `video1` while on Galaxy SII, it becomes `video0` for both front and rear (more details in IV-B). In the case of Galaxy GRAND, `vchiq` is its camera controller and `vc-cam` is its device node. Device files' group memberships may vary as well. For example, on Nexus 4, the camera device node is with `system:camera`, on Galaxy SII the group becomes `system:root` and on GRAND it turns out to be `system:system`. Vendors did not follow any guideline when configuring the Linux settings of their customized device files. With such a diversity, those device files were all identified by Device Miner.

LCFs detected. Those files were further correlated by Reference Identifier through their system call set and call arguments. As an example, Table VI shows the argument-level connections between camera device nodes on Nexus 4 and SII. By comparing the device-related files on the three customized phones with those on the reference, ADDICTED reported 4 LCFs, which are presented in Table V. Most problems come from obvious erroneous settings of sensitive device files to publicly accessible. We manually inspected those cases, which all look indeed problematic: the camera device nodes on SII and GRAND, the input device on SII, and the frame buffer on Galaxy ACE 3 are set to be publicly readable and writable. We further built end-to-end attacks to exploit one of these two camera flaws and the LCFs with the input device and the frame buffer to demonstrate the seriousness of the problem.

B. Attacks

To understand the seriousness of the LCFs we discovered, we thoroughly analyzed the 3 vulnerabilities and built end-to-end exploits on them. These vulnerabilities include the exposure of the input device node and camera device node on Galaxy SII, and the unprotected frame buffer on Galaxy ACE. We found that those flaws can be exploited by an app either without any permission or with unrelated ones such as `WRITE_EXTERNAl_STORAGE` (for the purpose of storing collected data). Some of such attacks can be quite complicated, due to the lack of documentations about those customized devices. We reported all our findings to Samsung and Google and are currently working with them to fix those flaws.

Touchscreen Keylogger. On Galaxy SII, ADDICTED discovered that part of the standard Linux input driver files (e.g., `/dev/input/event2`)[4] are made public. This problem was also found on a series of other phones, as reported by our measurement study (Section V). On Android, the input system is used for dispatching events to different services and delivering to them the data received by sensors like gyro and compass and other input hardware such as touchscreen and keyboard. Once the device files are exposed, any party can get from them sensitive sensor data and user inputs. In our research, we show that a touchscreen keylogger can be built by exploiting this vulnerability.

The Android input system includes three components, `EventHub`, `EventReader` and `EventDispatcher`. `EventReader` reads from different driver files (through `EventHub`) the data collected by sensors or input hardware and converts them into different events such as `KeyEvent`, `MotionEvent` etc. These events are dispatched to proper windows through `EventDispatcher`.

Now that all the device files are up for grabs, we implemented our own input reader to directly collect data from them, in the absence of proper permissions. Specifically, our reader is designed to work on touch events and capable of extracting the coordinates (`ABS_X`, `ABS_Y`) of any touch on the screen, together with its status (`ABS_PRESSURE` with 0 for up and 1 for down). The input reader can be embedded into a malicious app running in the background and silently logging all the touch events from the screen. Since a phone's keyboard layout is fixed, the app can easily identify all the keys the user enters from those events. The consequence of this attack is serious, which allows an unprivileged adversary to steal the phone user's password and any sensitive information she types through the touchscreen. Given the popularity of Galaxy SII and other phones with the same vulnerabilities, this problem affects millions of Samsung customers. A video demo of the attack [1], which shows the input reader recording coordinates of touch events, is posted online.

Camera attack. Our automatic analysis detected the exposure of the camera device node on Galaxy SII, a problem that also exists on other popular phones (Section V). Specifically, the file (`/dev/video0`) of Galaxy SII was correlated to the device node on the reference from their argument fingerprints, and it was also found to be public. With this device node's complete disclosure, one can take pictures without any camera-access permission in theory. However, exploiting this flaw in practice is highly nontrivial, as the driver is customized and Samsung never provides any documentation about how

[3]The two camera LCFs are similar. We just exploited one of them.

[4]The number may change depending on the systems and configurations.

TABLE V. LCFs DETECTED BY RISK IDENTIFIER

Device	Nexus 4			Galaxy SII			Galaxy ACE 3			Galaxy GRAND		
	/dev	mod	owner	/dev	mod	owner	/dev	mod	owner	/dev	mod	owner
camera	video0(rear)	660	system:camera	video0(front)	**666**	system:root	video0	660	system:camera	**vchiq**	**666**	system:system
	video1(front)	660	system:camera	video1(rear)	**666**	system:camera	video1	660	system:camera	**vc-cam**	**666**	system:system
input	input/event*	660	root:input	input/event*	**666**	root:input	input/event*	660	root:input	input/event*	660	root:input
framebuffer	graphics/fb*	660	root:graphics	graphics/fb*	660	root:graphics	graphics/fb*	**666**	system:graphics	graphics/fb*	660	root:graphics

to directly work on it. Following we describe our end-to-end attack on this vulnerability.

To communicate with the camera device, we first tried the operation sequence specified by the V4L2 standard [17] as shown on the left column of Table VI. This approach turned out to be ineffective and keep crashing and rebooting the phone when we invoked the command `VIDIOC_ENUM_FMT` through `ioctl`, which itself is a denial-of-service attack. To find out the correct way to take a picture, we first analyzed the system-call traces collected from normal operations of the camera on SII and found that the sequence here is different from that on Nexus 4. Specifically, we found that in Samsung SII, `video0` is for both its front and rear camera and a camera client needs to select the front or rear channel through `VIDEO_S_INPUT` before using the camera. Even after we made the sequence right, our app still caused the system to crash at `VIDIOC_S_FMT`, a command for specifying the format parameters for pictures. By digging the specifications for the camera chips used in the phone (S5K5BAFX and M5M0), we finally realized that Samsung customized the V4L2 protocol defined in `videodev2.h` with new settings. For example, the V4L2 standard buffer type `v4l2_buf_type` has been extended by adding `V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLANE`, `V4L2_BUF_TYPE_VIDEO_OUTPUT_MPLANE` and `V4L2_BUF_TYPE_PRIVATE`. We ended up implementing the whole Hardware Abstract Layer (HAL) within our app, based upon an open-source driver for the same chips [15].

Using our own HAL, we successfully commanded our attack app (without any camera permission) to open the camera, take pictures, retrieve camera raw data from the exposed device and covert them from the YUV color space [18] to the RGB color space [14] for constructing images.

TABLE VI. DIFFERENCE OF CAMERA OPERATIONS ON NEXUS 4 AND SAMSUNG SII

Nexus 4	Samsung SII	Operations
`open`	`open`	open camera device
VIDIOC_QUERYCAP	VIDIOC_QUERYCAP	query capabilities
	VIDIOC_ENUMINPUT	enumerate video input
	VIDIOC_S_INPUT	set the input
	VIDIOC_S_CTRL*	set white balance, mode, focus
VIDIOC_ENUM_FMT	VIDIOC_ENUM_FMT	enumerate supported format
VIDIOC_S_FMT	VIDIOC_S_FMT	set format
VIDIOC_REQBUFS	VIDIOC_REQBUFS	allocate shared memory
VIDIOC_QUERYBUF	VIDIOC_QUERYBUF	query the status of the buffer
VIDIOC_QBUF	VIDIOC_QBUF	exchange a buffer with the driver
VIDIOC_STREAMON	VIDIOC_STREAMON	start or stop streaming I/O
VIDIOC_DQBUF	VIDIOC_DQBUF	exchange a buffer with the driver
	VIDIOC_S_CTRL	pause in our case
	VIDIOC_STREAMOFF	close the stream

Again, our attack does not need any camera-access permissions, and nor does it demonstrate any visual effects during picture taking (such as showing preview in a normal use of the camera). Therefore, it is completely stealthy to the phone user. A video demo of the attack is here [1]. This problem affects millions of phone users (Section V).

Screenshot capture. Screenshot taking is considered to be a highly sensitive capability, which Google guards with a system permission `READ_FRAME_BUFFER` and never grants to any third-party app. Actually, apps that provide programmatic approaches to capturing screenshots need to either run on a rooted phone or get the capability through Android Development Bridge [10]. However, we found in our study that some vendors expose this capability to the public. Specifically, on Galaxy ACE 3 GT-7270L (running 4.2.2), the Linux standard frame buffer device `/dev/graphics/fb0` is set to be publicly readable and writable. The `fb0` file is a hardware independent graphic abstraction layer and contains the current image on screen. We suspect that this oversight (exposure of the frame buffer) could be caused by the attempt to provide a customized screenshot capability for vendor pre-installed apps such as screenshot for Samsung Note app. In our study, we implemented an attack on this flaw over Samsung Galaxy ACE 3, a phone model distributed in Latin America.

In the attack, again we implemented a malicious app that runs as a background service and periodically reads from the frame buffer. Each time, the app converts the content it gets from the buffer into an image file (JPG) and saves it on the phone's SD card. Whenever the user is running sensitive apps such as those providing financial services (which can be found out from the package names of the running processes), the malicious app continuously takes screenshots to collect sensitive user data. The demo of this attack is also online [1].

All those malicious apps can be made highly context-aware: that is, they can continuously monitor what the phone user is running and only start collecting information at the right moment. This can be done through inspecting the list of running processes and CPU, memory usages of those apps, as proposed in prior research [46].

V. A LARGE-SCALE MEASUREMENT STUDY

Our study on 4 Android devices, as described in Section IV, reveals the great impact of customization errors to Android device security, which allow an unauthorized app to get access to critical system capabilities on popular phones. What is less clear is how pervasive those security-critical flaws are across a large number of phones, tablets and all kinds of Linux devices, beyond those on our short list (Table V). Since it is hard to get the answer to this question by running ADDICTED over thousands of physical devices, we have to statically check factory images. The problem is that without hardware, we cannot perform the dynamic analysis to identify all customized

TABLE VII. ANDROID FACTORY IMAGES COLLECTED BY OUR CRAWLER

Version	# of Phone Models of each Version	# of Images
4.0.3	16	109
4.0.4	113	564
4.1.1	48	238
4.1.2	159	1054
4.2.1	1	1
4.2.2	59	397
4.3	21	60
TOTAL	417 (288 distinct models)	2423

device files even for those on our short list. All we can do is to use the names of the device related files on the reference to search for those on customized images to find out whether they are as well protected there as on the reference.

Also we want to go beyond the devices on the list. Device nodes are typically located under `/dev` with some of them placed under `/sys`. Finding what hardware pieces those individual files serve is challenging. In our research, we just blindly compared them with their counterparts on the reference according to their names, to get some clues about whether they are well configured based on their Linux permission settings. Another trouble here is that `/sys` and `/dev` are actually dynamically generated by the Linux `init` process. To find out those virtual files' security settings, we resorted to the configuration files that `init` utilizes to determine their protection levels. Specifically, when Linux initializes, `init` runs the `ueventd` process to read from `ueventd.rc` and `ueventd.$HARDWARE.rc` settings of different device nodes. In our research, we just extracted these two files from factory images and used them to analyze device-related files' Linux protection. Following we first explain the methodology used in our study and then present the findings we made.

A. Methodology and Data

As discussed above, the methodology of our measurement study is to analyze the configuration files of different factory images, comparing the device settings on these images (as recorded in the files) with those on the reference to identify LCFs. Most important to this study is to find out as many images as possible to collect their configuration files. Here we discuss how this was done in our research.

Factory image collection. Although vendors release the source code of their Android updates, they typically do not make public the configuration files we are looking for. Those files can only be extracted from factory images. In our study, we developed a crawler that automatically checks top image storing websites, including Samsung update [5] and full-firmwarec [4], to download those images. From all the images there, we selected one image for each phone model each version (at least 4.0). In total, our crawler gathered 2423 images covering 417 phone models ranging from android 4.0 to 4.3 and total of 2.5TB disk size. Table VII summarizes the scale of our study. (Note that one phone model even at one specific version may have multiple images for different countries or regions.)

Configuration file extraction. The factory image we downloaded is usually in zip or tar format. To extract from it the configuration files, our approach first unzipped the image to search for `zImage` or `boot.img`. Then it ran zImageTool and Boot-Image-tools [9] to split the image into kernel and ramdisk. From the ramdisk part, we retrieved the `.rc` configuration files. Those files were parsed and their relevant content was stored into a database. During this process, `ueventd.$HARDWARE.rc` was always processed after `ueventd.rc`, as the configurations in the former overrule those in the latter during Linux initialization.

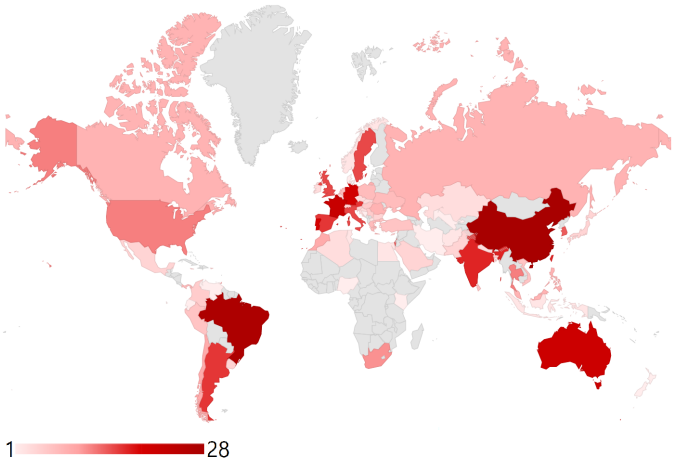

Fig. 3. Number of phone models with the 3 confirmed vulnerabilities. No data is available for gray areas.

All the file settings from one image were then compared with those on the reference with the image's Android version. For example, the two configuration files from a Samsung customized 4.0.4 were analyzed against those on AOSP 4.0.4.

B. Results

Pervasiveness of LCFs. Our study shows that security hazards of device customizations are indeed pervasive. Table VIII summarizes the total number of LCFs we found. From the table, we can see that 1290 (53.24%) images we studied contain LCFs. More specifically, they include at least one device file whose protection level is set to be publicly readable and writable, way below that of the same file on the reference. We found that such LCFs affect 75.65% of the distinct phone models and 86.65% of the carriers we studied.

We also measured the magnitude of the confirmed vulnerabilities we exploited (Section IV). Table VIII shows the pervasiveness of these flaws across different customized devices. Just for an example, the problem with the camera driver was found within 952 images for 72 phone models.

Distribution of the flaws. We further studied the distribution of the LCFs across different geo-locations. For this purpose, we identified the countries of different factory images and mapped to them the number of the phone models in those individual countries affected by the 3 confirmed flaws (Section IV-B). Figure 3 illustrates the results, which are presented on the Google maps. Interestingly, we found that developing countries (e.g., China, Brazil) host more vulnerable phone models. The security quality of customizations there may be lower than that in other countries.

TABLE VIII. IDENTIFIED LCFS AND CONFIRMED VULNERABILITIES

	# of distinct device nodes (Total on references: 222)	# of distinct images (Total: 2423)	# of distinct models (Total: 288)	# of carriers (Total: 275)
LCFs	28(12.61%)	1290(53.24%)	215(75.65%)	238(86.55%)
INPUT	1	329(13.58%)	35(12.15%)	136(49.45%)
VIDEO	5	952(39.23%)	72(22.00%)	217(78.91%)
FRAME_BUFFER	1	90(3.71%)	14(4.86%)	47(17.09%)

TABLE IX. TOP 10 REGIONS WITH MOST LCFS AND CONFIRMED VULNERABILITIES

RANK	Region	# of models with LCFs	# of models with the 3 vulnerabilities
1	CHINA	80	28
2	BRAZIL	65	27
3	FRANCE	51	22
4	US	46	12
5	INDIA	42	17
6	GERMANY	41	20
7	TAIWAN	40	17
8	SPAIN	40	16
9	HONG KONG	38	14
10	AUSTRALIA	37	21

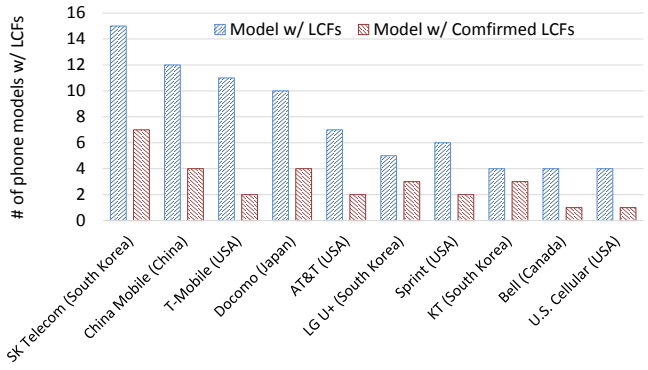

Fig. 4. Top 10 Carriers that has most LCFs and Confirmed Vulnerabilities.

We also analyzed the distributions of all LCFs discovered through the comparison between factory images and references. Table IX presents the top 10 countries in terms of the number of phone models involving LCFs and the number of the models with the 3 confirmed vulnerabilities. As we can see here, these two ranking lists are largely aligned.

Besides vendors, carriers may also customize phones to add new features. Figure 4 shows the top 10 carriers with the highest number of phone models affected by the LCFs and confirmed vulnerabilities. Among them, the SK Telecom (Korea) and China Mobile have the largest number of vulnerable phone models. T-Mobile and AT&T are also on the list. Particularly, they have few popular models (Galaxy Note 2 and Galaxy Tab for T-Mobile, and Galaxy SII for AT&T) that contain the confirmed customization vulnerabilities described in Section IV.

To find out how the security quality of customizations evolves across different Android versions, we inspected the factory images across multiple Android versions for the presence of LCFs. Table X shows the number of LCF-affected phone models on different versions. When those phones are upgraded from 4.0.3 to 4.1, the percentage of vulnerable models drops from nearly 70% to 18%. However, it goes up to round 30% for 4.1.2 and 4.3. This shows that vendors seem to stop making much progress on suppressing such customization flaws.

TABLE X. DISTRIBUTION OF LCFS OVER OS VERSIONS

OS Version	Phone Models with LCFs	Total Phone Models	%
4.0.3	11	16	68.75%
4.0.4	78	113	69.03%
4.1.1	9	48	18.75%
4.1.2	48	159	30.19%
4.2.2	14	59	23.73%
4.3	6	21	28.57%

We further selected all 103 phone models with at least 2 images (of course, with different Android versions) to study the evolution of LCFs across different versions on individual models. Out of them, 92 phone models have at least one of the LCFs, including the 3 confirmed vulnerabilities, on at least one of their images. On these 92 models, we checked whether their LCFs are removed after the phones are upgraded. Table XI shows the number of phone models whose updates fix their existing LCFs or introduce new ones. As we can see here, only on 6 phone models, the vendors have completely addressed all their LCFs through updates, while the others either continue to suffer from at least one of the existing LCFs, or even get new ones. The most interesting case is GT-N7000 Galaxy Note: its camera vulnerability (Section IV-B) is present on both 4.0.3 and 4.0.4; the problem disappears on 4.1.1 but shows up again on the 4.1.2. Therefore, we are not sure whether the vendor has discovered the problem, intentionally addressed it and made the same lapse again later. Also for the frame buffer flaw, we only have images for the affected phones at 4.1 and 4.2 and therefore do not know whether they have been addressed in their most recent versions.

TABLE XI. LCFS FIXED OR INTRODUCED BY UPDATES ON PHONE MODELS

	# of Phone Models
Fixed at least 1 LCF when upgrading	53
Fixed all LCFs on the phone when upgrading	6
Fixed LCF shows up again after upgrades	2
Introduce new LCFs when upgrading	36

Popular public device files. Finally, we took a look at all public device files under `/dev`. Table XII lists top device files based upon their individual number of occurrences across various customized phones. On the list, only the first one `/dev/kgsl-2d1` was reported to have an LCF, which is a driver for a 2D graphic acceleration card. Its disclosure to the public could allow an unauthorized party to access the information about the images displayed on the phone's screen, though whether this exploit can work out needs a further investigation. Other device files on the list do not even show up on the references, so we did not have any indications that they need to be protected. To find out what those files are, we manually analyzed their source code and searched for their information on Google. It turns out that

TABLE XII. MOST POPULAR PUBLICLY ACCESSIBLE DEVICE NODES AND THEIR POTENTIAL IMPACTS

Device Node	Functionality and Impact
/dev/kgsl-2d1	GPU device for 2D acceleration
/dev/ump	Unified Memory Provider that provides a way to share both existing and new memory areas across processes and hardware units.
/dev/fimg2d	2D graphic driver that provides image transformation for stretching, rotation and alpha blending. Data will be delivered to framebuffer.
/dev/hwmem	An interface to allocate contiguous memory buffers for hardware and handle sharing of allocated buffers between processes
/dev/s5p-mfc	Camera driver. Exposure of this device could lead to camera attacks.
/dev/exynos-mem	Memory used by graphics, surfaceflinger. Exposure of this device could enable an app to tamper kernel memory.
/dev/vc-lmk	Low memory kill. Exposure of this device may allow an ordinary app to kill any process.
/dev/felica	Driver for RFID smart card system.

all of them are indeed security-critical. For example, the file `/dev/vc-lmk` was found to be a "low-memory kill" device, which terminates an app when the phone's memory is running low. With its complete exposure, *any* app, without any permission, can invoke the device to stop another app. Another example is `/dev/hwmem`, a memory device through which one can touch physical memory. This device has been made public on Galaxy S 3 Mini and other models. As a result, an unprivileged app may be able to make `ioctl` calls to this device to read or write part of the phone's physical memory. All such findings, again, show that what we detected through ADDICTED are nothing more than a tip of the iceberg. The security problems in Android device customizations are so serious that immediate efforts need to be made to better understand them and effectively address them.

More attacks. We further analyzed two potential vulnerabilities in Table XII and developed end-to-end attacks on them. This confirmed that those flaws are real and their exploits can have serious consequences. Specifically, we studied the `vc-lmk` device, which, as discussed before, is a low memory kill driver. Android is designed to accommodate as many processes as possible in the memory to help them promptly respond to the user's requests. However, when the memory is about to run out, the OS picks up processes, based upon their `oom_adj` values, to terminate. For this purpose, Android includes this process-killing device, which should never be made public. In our research, we implemented an app, without any permission, to exploit this exposure to terminate other processes. Since the exposed driver runs in the kernel land, it can stop any processes, including system apps like Phone, `SurfaceFlinger` and even the `init` process. To command the driver, the app was built to make `ioctl` calls to `vc-lmk` with the command `VC_LMK_IOC_KILL_PID` and `pid` as the argument. Executing the command, the driver then sends an `SIGKILL` to the target and asks it to stop running. Note that there is no access-control protection whatsoever within `ioctl` to prevent our app from unleashing `vc-lmk`. In our experiment, we successfully stopped several system processes. Interestingly, once the Phone app is terminated, ongoing calls hang up; `SurfaceFlinger`'s termination immediately freezes the phone's user interface; When the `init` process is stopped, the whole phone reboots.

We also came up with an attack on the exposed `UMP` device, which is a unified memory resource allocator. Our analysis of its source code reveals two IO control commands: `UMP_IOC_ALLOCATE` for allocating memory and copying user data to kernel, and `UMP_IOC_MSYNC` for cache maintenance. Our attack leveraged the first command to continuously require resources until disabling the phone by using up all it memory. For the second command, we ran it to access a random memory location, which caused the phone to reboot immediately. Clearly, the device provides security-critical capability and therefore should not be made available to unauthorized apps.

VI. DISCUSSION

The openness of Android has brought in a fragmented ecosystem with significant security implications. In our research, we made the first step toward understanding the security challenges there, particularly the security-critical flaws introduced during customization of Android Linux devices. We found the presence of customization errors in the security configurations of device-related files on a large number of Android phones, across multiple OS versions, different vendors, carriers and regions, leading to complete exposure of critical system resources and capabilities. On the other hand, we believe that what we found just scratches the surface of the new security challenges in device customizations. Further research effort is urgently needed on the following directions.

Other devices. Our techniques cannot connect most device-related files to their Linux devices. Under `/dev`, still dozens of files are there which we cannot interpret, though some of them may not be sensitive. To increase the coverage, more test cases need to be added to ADDICTED's test suite. More challenging here is identification of the drivers for the devices not supported by the official Android OSes. A comparison with the reference in this case will not help detect the security-critical flaws in their device files. New techniques therefore need to be developed to address this problem.

Flaw detection. The design of ADDICTED is to detect LCFs by looking at the way device files are protected on the reference. For those with downgraded protection levels, all we can say here is that they might cause some security concerns. Confirmation of security flaws still need a manual analysis. New techniques for automating this process are definitely valuable. Also, ADDICTED is not designed to identify the device file configuration problems on the official Android OSes. Further research on this direction could leverage observed relations between Android permission levels and their corresponding devices' Linux-level protection settings to build a model for detecting security configuration flaws on AOSP versions.

VII. RELATED WORK

In this section, we review related prior studies and compare them with our work.

Android app analysis. Android apps have often been analyzed statistically for malware detection [47] and vulnerability identification. Examples include CHEX [38] and ComDroid [23] that utilize this technique to find out security-critical flaws within apps. Also, Woodpecker [30] scans a large number of

apps for their privilege-leak weaknesses. Different from those prior approach, ADDICTED is meant to work on the Android system, instead of individual apps. Given the complexity of the system, which includes both C code and Java programs, tracking operations on device files is hard to be done statically.

Most relevant to our work is the recent research on vendor customization of pre-installed apps [43]. This research statically analyzes 10 representative stock Android images from 5 top vendors to identify the provenances of their pre-installed apps. It further discovers that a large portion of those apps are overly privileged or have re-delegation vulnerabilities, and also most of the problematic apps come from the vendors. Unlike this prior work, whose focus is the impact of customizations on apps, our work looks at device drivers, which are the predominant cause for phone vendors to customize Android. What we found, including the pervasiveness of exposed Linux device nodes and the serous consequences once they are exploited, have never been reported before.

When a security analysis on Android needs to touch its system code, oftentimes, this needs to be done through a dynamic analysis [2, 26, 32, 44]. A prominent example of this line of work is TaintDroid [26], a tool designed for dynamic taint analysis on Android. This approach can achieve a fine-grained tracking of data flows across apps and the OS. However, it is less suitable for analyzing multiple customized systems because it requires an intensive instrumentation of the OS to get a good performance, which limits its portability. Another fine-grained dynamic tool is Droidscope [44], which runs the whole Android platform on an emulator to reconstruct both the OS and Dalvik level views of the system. The problem is that it is difficult for the emulator to mimic different customized hardware, which is required for our study on Android Linux devices.

System-call analysis. Given the challenges of using existing tools for our study, we built our own dynamic analysis tool that works on the system-call level to achieve a high performance and portability. System calls have long been used for security-related program analysis [21, 36, 37, 40]. ADDICTED is built on strace that is ported and extended to work on ARM-based systems.

Android permissions. Also related to our research is a large amount of the literature on Android permissions. Those permissions are meant to protect critical Android resources on the framework layer [24, 27–29]. Prior work leverages dynamic analysis to "demystify" Android permissions, mapping Android APIs to their related permissions [28]. The outcomes of the study can help us find the right APIs to trigger device-related permissions, for the purpose of locating the Linux files related to the device. Another example is the technique for tracking an IPC call's provenance to prevent permission re-delegation attacks. Our approach also monitors the IPC but for finding the file-system activities in response to the request. There is also a line of research on enhancement of the permission system [19, 22, 25, 33, 35, 39]. Fundamentally, all those prior studies focus on the security protection on the framework layer, while our research investigated the Linux-layer security hazards introduced by device customizations.

Android Linux-layer security. Only limited effort has been made on Android's Linux-layer security. Prominent examples include Momento [34], which investigates the information leaks from the shared memory usage data exposed by Android's Linux, and the recent work on Android public information leaks [46]. Those studies follow the foot step of the work on the privacy implications of the Linux Proc file system [45]. Different from the prior work, our research investigates the Linux-layer protection of Android device files, whose exposures have direct and often more serious consequences, as shown in the paper. Also, effort has been made recently to enhance access control on Android [42], based upon SELinux [6].

Sensor data inference. There is a line of research on inferring sensitive user information from the public data exposed by Android devices, particularly the outputs of different sensors [20, 31, 41]. Different from such work, our approach reveals the customization errors that cause explicit disclosure of Android device nodes, allowing an unauthorized party to directly get information from them.

VIII. Conclusion

The fragmentation of the Android ecosystem has brought in new security challenges: vendors and carriers aggressively customize official OS versions to accommodate their new hardware pieces and services, which can potentially undermine Android security protection. This important issue, however, has not been adequately studied. Particularly, little is known about the security implications of customizing a variety of Android Linux devices such as camera, audio, GPS, etc. In our research, we made the first step toward better understanding of this issue, leveraging a new technique, ADDICTED, designed for automatic detection of some types of security-critical customization flaws. ADDICTED dynamically analyzes the operations on a sensitive Android device to connect it to a set of Linux device files. The security protection of these files is then evaluated against that received by their counterparts on the AOSP OS. In this way, our approach automatically identifies those under-protected device nodes. Running ADDICTED on popular phone models, we discovered critical flaws that allow an unauthorized app to take pictures and screenshots, and even record the user's input keys from touchscreen. Those vulnerabilities were found to exist on hundreds of other phone models. Our measurement study further reveals the LCFs present in over 1,000 phone models distributed across different Android versions, carriers and countries.

With the important discoveries we made, our research just scratches the surface of the grand security challenges that come with Android customizations. Even on the Linux layer, still there are many device files we cannot interpret, not to mention detection of their security flaws. More importantly, further effort is expected to understand how to protect security-critical resources on different Android layers, and develop effective means to ensure that customized resources are still well guarded.

Acknowledgements

The project was supported in part by the NSF CNS-1017782, 1117106, 1223477 and 1223495.

REFERENCES

[1] Demo of the paper. https://sites.google.com/site/linuxdroid0/.

[2] Droidbox: Android application sandbox. https://code.google.com/p/droidbox/. Accessed:Nov, 2013.

[3] File, folder comparison & synchronization for mac os x. http://www.deltopia.com/compare-merge-sync/macosx/. Accessed: 05/20/2013.

[4] Full firmware. http://www.full-firmware.com/. Accessed:05/02/2013.

[5] Samsung updates: Latest news and firmware for your samsung devices! http://samsung-updates.com/. Accessed: 05/02/2013.

[6] Se linux. http://www.nsa.gov/research/_files/selinux/papers/slinux.pdf. Accessed: 11/09/2013.

[7] Using the input subsystem. http://www.linuxjournal.com/article/6429, year = 2013,.

[8] Android tops 81 percent of smartphone market share in q3. http://www.engadget.com/2013/10/31/strategy-analytics-q3-2013-phone-share/, 2013. Accessed: 10/31/2013.

[9] Boot image tools. https://github.com/sakindia123/Boot-Image-tools, 2013.

[10] How to take screenshots on your unrooted android phone – windows version. http://www.lindylabs.com/screenshot_it/instructions_win.html, 2013.

[11] Linux-fbdev.org. http://www.linux-fbdev.org/, 2013.

[12] List of best-selling mobile phones. http://en.wikipedia.org/wiki/List_of_best-selling_mobile_phones, 2013.

[13] randoop, random test generation. https://code.google.com/p/randoop/, 2013. Accessed: 11/08/2013.

[14] Rgb color model. http://en.wikipedia.org/wiki/RGB_color_model, 2013.

[15] Samsung s5p/exynos4 fimc driver. https://www.kernel.org/doc/Documentation/video4linux/fimc.txt, 2013.

[16] strace. http://sourceforge.net/projects/strace/, 2013. Accessed: 11/08/2013.

[17] V4l2 framework. https://www.kernel.org/doc/Documentation/video4linux/v4l2-framework.txt, 2013.

[18] Yuv. http://en.wikipedia.org/wiki/YUV, 2013.

[19] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM.

[20] Liang Cai and Hao Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX conference on Hot topics in security*, HotSec'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[21] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 122–132, New York, NY, USA, 2012. ACM.

[22] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *NDSS*. The Internet Society, 2013.

[23] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[24] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, August 2011.

[25] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS)*, February 2011.

[26] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[27] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM CCS*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[28] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[29] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, pages 22–37, 2011.

[30] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, February 2012.

[31] Jun Han, Emmanuel Owusu, Thanh-Le Nguyen, Adrian Perrig, and Joy Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Proceedings of the 4th International Conference on Communication Systems and Networks*, Bangalore, India, 2012.

[32] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM.

[33] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM CCS*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[34] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the*

2012 IEEE Symposium on Security and Privacy, SP '12, pages 143–157, Washington, DC, USA, 2012. IEEE Computer Society.

[35] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka 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*, pages 775–792. Springer, September 2013.

[36] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 351–366, Berkeley, CA, USA, 2009. USENIX Association.

[37] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 399–412, New York, NY, USA, 2010. ACM.

[38] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[39] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[40] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6^{th} European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.

[41] Roman Schlegel, Kehuan Zhang, Xiao yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society, 2011.

[42] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*. The Internet Society, 2013.

[43] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security*, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[44] Lok Kwong Yan and Heng Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[45] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 17–32, Berkeley, CA, USA, 2009. USENIX Association.

[46] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of 20th ACM Conference on Computer and Communications Security (CCS)*, November 2013.

[47] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian 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*, February 2012.