Security Threats to Android through Vendor Customization

Kailash Khatwani[1]
[1] Dept. Of MCA
[1] Vivekanand Education Society Institute of Technology, Affiliated to Mumbai University, Mumbai, India.

Abstract- *Android's popularity is due in part to it being an open platform. Google produces a baseline version of Android, and then makes it freely available in the form of the Android Open Source Project (AOSP). Manufacturers and carriers are free to build upon this baseline, adding custom features in a bid to differentiate their products from their competitors. These customizations have grown increasingly sophisticated over time, as the hardware has grown more capable and the vendors more adept at working with the Android framework. Flagship devices today often offer a substantially different look and feel, along with a plethora of pre-loaded third-party apps. From another perspective, vendor customizations will inherently impact overall Android security. Past works have shown that Android devices had security flaws shipped in their preloaded apps. Note that stock images include code from potentially many sources: the AOSP[34] itself, the vendor, and any third-party apps that are bundled by the vendor or carrier. It is therefore important to attribute each particular security issue back to its source for possible bug-fixes or improvements. We aim to study vendor customizations on stock Android devices and assess the impact on overall Android security.*

Keywords- android, analysis, security, vendor, customization ,analysis, smartphones, threats.

I. INTRODUCTION

Android is an open-source Operating System created for mobile phones and other devices led by Google. In 2003, Andy Rubin and his team invented Android [1]. Later in 2005, Google acquired the Android operating system. Android is a software environment built for mobile devices. It is not a hardware platform. Android includes a Linux kernel-based OS, a rich UI, end-user applications, code libraries, application frameworks, multimedia support, and much more [1]. On the 5th of November in 2007, distribution of Google's version was announced with the founding of the Open Handset Alliance. Open Handset Alliance is a group of device manufactures, Chipset Manufacturers and Mobile Carriers. Device Manufacturers include companies like HTC, LG, Samsung, Motorola and Sony. Chipset makers are Qualcomm, Texas Instrumental and Intel. Mobile Carrier companies are represented by companies like Verizon, Sprint, AT&T, T-Mobile, MetroPCS and lots of other companies all over the world [2]. Most of the Android platform is released under Apache 2.0 license. Google Android released the entire source code under an Apache license [1]. With Apache license, a user can freely download and use Android for personal and commercial purposes. It allows user to make changes to original software without having to contribute to the open source community.

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 March 2014, Android has dominated global smartphone shipments with nearly 80% market share. 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. 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. Most customizations on the Android kernel layer are actually related to those devices, 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[35]). 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.To that end, we perform a three-stage process to evaluate a given smartphone's stock firmware image. First, we perform provenance analysis, aiming to classify each pre-loaded app into three categories:

1. apps originating from the AOSP[34]
2. apps customized or written by the vendor, and
3. third-party apps that are simply bundled into the stock image.

We then analyze, in two different ways, the security implications of each app:

(1) Permission usage analysis compares the permissions requested by the app with those that it actually uses, looking for apps that request more permissions than they use. This situation is known as permission over privilege, and it indicates a poor understanding of the Android security model

(2) Vulnerability analysis, in comparison, looks for two general types of actual security vulnerabilities: permission re-delegation attacks and content leaks. Permission re-delegation attacks allow unprivileged apps to act as though they have certain sensitive permissions, while content leaks allow such apps to gain (unauthorized) access to private data.

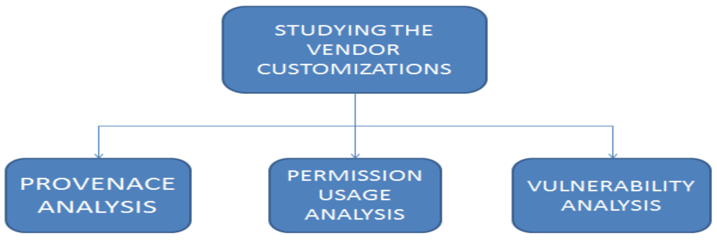

Figure 1. Methodology of Analysis.

II. PROBLEM DEFINITION

The smartphone market has grown explosively in recent years, as more and more consumers are attracted to the sensor-studded multipurpose devices. Android is particularly ascendant; as an open platform, smartphone manufacturers are free to extend and modify it, allowing them to differentiate themselves from their competitors. However, vendor customizations will inherently impact overall Android security and such impact is still largely unknown. When Android phone manufacturers tweak devices and customize phones with special software, apps and code, it has a direct effect on the security of each device. In our study, we analyze six representative stock Android images from popular smartphone vendors. Our goal is to assess the extent of security issues that may be introduced from vendor customizations and further determine how the situation is evolving over time. In particular, we take a three-stage process: First, given a smartphone's stock image, we perform provenance analysis[36] to classify each app in the image into three categories: apps originating from the AOSP, apps Customized or written by the vendor, and third-party apps that are simply bundled into the stock image. Such provenance analysis allows for proper attribution of detected security issues in the examined Android images. Second, we analyze permission usages of pre-loaded apps to identify overprivileged ones that unnecessarily request more Android permissions than they actually use. Finally, in vulnerability analysis, we detect buggy pre-loaded apps that can be exploited to mount permission re-delegation attacks or leak private information. Our evaluation results are worrisome: vendor customizations are significant on stock Android devices and on the whole responsible for the bulk of the security problems we detected in each device. Specifically, our results show that on average 83.15% of all preloaded apps in examined stock images are overprivileged with a majority of them directly from vendor customizations. In addition, 64.71% to 85.00% of vulnerabilities we detected in examined images from every vendor arose from vendor customizations. In general, this pattern held over time – newer smartphones, we found, are not necessarily more secure than older ones.

III. FACTS AND FINDINGS

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, Airtel 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 researches show 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[37] (Unified Memory Provider) driver to allocate memory; for this purpose, Samsung made UMP publicly accessible. Making the UMP accessible publicly compromises with the security of the device.

IV. AUTOMATED COLLECTION TOOLS

The goal of this research is to study vendor customizations on stock Android devices and assess corresponding security impact. Note that the software stack running in these devices are complex, and their firmware is essentially a collaborative effort, rather than the work of a single vendor. Therefore, we need to categorize the code contained in a stock image based on its authorship and audit it for possible security issues. After that, we can attribute the findings of the security analyses to the responsible party, allowing us to better understand the state of smartphone security practices in the industry and spot any evident trends over time. Following three technologies were briefly used in our research:

1. ES File Manager File Explorer
2. Android-apktool
3. ASEF (Android Security Evaluation Framework)

Out of these the firt two are simple and widely used in the Android domain.The third,however, deserves a little enlightenment as will be evident in the succeeding disussion:
ASEF is designed and developed to simulate the entire lifecycle of an Android application in an automated virtual environment to collect behavioral data and perform security evaluations automatically over 'n' number of apps. Android Security Evaluation Framework (ASEF) performs this analysis while alerting you about other possible issues. It will make you aware of unusual activities of your apps, will expose vulnerable components and help narrow down suspicious apps for further manual research. The framework will take a set of apps (either pre-installed on a device or as individual APK files) and migrate them to the test suite where it will run it through test cycles on a pre-configured Android Virtual Device (AVD).During the test cycles the apps will be installed and launched on the AVD. ASEF will trigger certain behaviors by sending random or custom gestures and later uninstall the app automatically. It will capture log events, network traffic, kernel logs, memory dump, running processes and other parameters at every stage which will later be utilized by the ASEF analyzer. The analyzer will try to determine the aggressive bandwidth usage, interaction with any command

and control (C&C) servers using Google's safe browsing API, permission mappings and known security flaws. ASEF can easily be integrated with other open source tools to capture sensitive information, such as SIM cards, phone numbers and others.ASEF is an Open Source tool for scanning Android Devices for security evaluation. Users will gain access to security aspects of android apps by using this tool with its default settings. An advanced user can fine-tune this, expand upon this idea by easily integrating more test scenarios, or even find patterns out of the data it already collects. ASEF will provide automated application testing and facilitate a plug and play kind of environment to keep up with the dynamic field of Android Security.

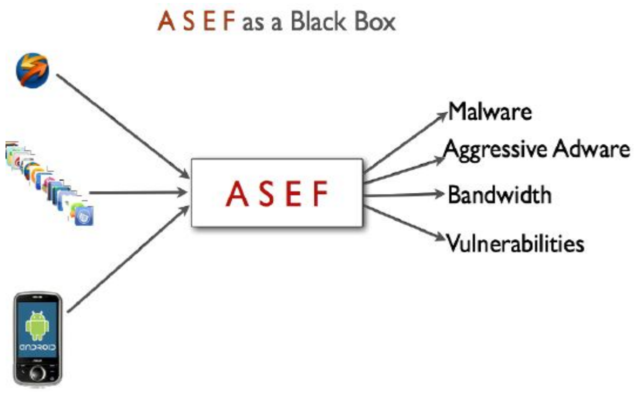

Figure 2. The ASEF Framework

V. RESEARCH METHOLOGY

The overall architecture of the ASEF system is explained in the following figure. Our system takes a stock phone image as its input, preprocessing each app and importing the results into a database. This database, initially populated with a rich set of information about pre-loaded apps (including information from their manifest files, signing certificates, as well as their code, etc.), is then used by a set of subsequent analyses. Each analysis reads from the database, performs its analysis, and stores its findings in the database. To study the impact of vendor customizations on the security of stock Android smartphones, we performed three such analyses.

First, to classify each app based on its presumed authorship, we perform provenance analysis. This analysis is helpful to measure how much of the baseline AOSP is still retained and how much customizations have been made to include vendor specific features or third-party apps.

further get a sense of the security and privacy problems posed by each app, we use two different analyses: Permission usage analysis[40] assesses whether an app requests more permissions than it uses, while Vulnerability analysis scans the entire image for concrete security vulnerabilities that could compromise the device and cause damage to the user. Ultimately, by correlating the results of the security analyses with the provenance information we collected, we can effectively measure the impact of vendor customizations.

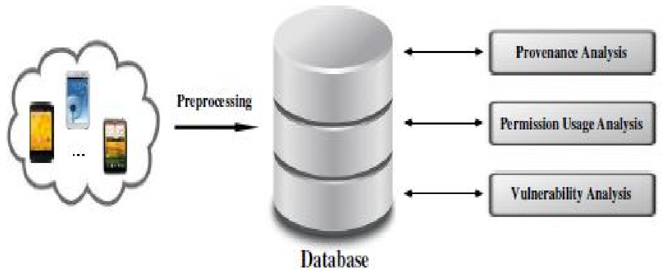

Figure 3. RESEARCH METHOLOGY

VI. PROVENANCE ANALYSIS

The Main Purpose Of Provenance Analysis Is To Study The Distribution Of Pre-Loaded Apps And Better Understand The Customization Level By Vendors On Stock Devices. Specifically, We Classify Preloaded Apps Into Three Categories:

- AOSP app: the first category contains apps that exist in the AOSP and may (or may not) be customized by the vendor.
- Vendor app: the second category contains apps that do not exist in the AOSP and were developed by the vendor.
- Third-party app: the last category contains apps that do not exist in the AOSP and were not developed by the vendor.

The idea to classify pre-loaded apps into the above three categories is as follows.

First we collect AOSP app candidates by searching the AOSP, then we exclude these AOSP apps from the pre-loaded ones. After that, we can classify the remaining apps by examining their signatures (i.e., information in their certificate files) based on a basic assumption: third-party apps shall be private and will not be modified by vendors. Therefore, they will not share the same signing certificates with vendor apps. In practice, this process is however not trivial. Since AOSP[34] apps may well be customized by vendors, their signatures are likely to be changed as well. Although in many cases, the app names, package names or component names are unchanged, there do exist exceptions. For example, Sony's Conversations app, with package name com.sonyericsson.conversations, is actually a customized version of the AOSP Mms app named com.android.mms. In order to solve this problem, we perform a call graph similarity analysis, which has been demonstrated to be an effective technique even to assist malware clustering and provenance identification. To generate the call graph required by any such analysis, we add all method calls that can be reached starting from any entrypoint method accessible to other apps or the framework itself. However, we are hesitant to use graph isomorphism techniques to compare these call graphs, as they are complex and have undesirable performance characteristics. Instead, we notice that later analysis will use paths, sequences of methods that start at an entrypoint and flow into a sink (i.e., API[39] or field which may require sensitive permissions, lead to dangerous operations or meet other special needs). Therefore, we choose to preprocess each app, extract and compare the resulting paths, a much more straightforward process that still compare the parts of each app that we are most concerned with.

From our prototype, we observe that such a path-based similarity analysis is implementation-friendly and effective. Particularly, we use the return type and parameters (number, position and type) of each method in the path as its signature. If the similarity between two paths exceeds a certain threshold, we consider these two paths are matching. And the similarity between two apps is largely measured based on the number of matched paths. In our prototype, to determine which apps belong to the AOSP, we accordingly take the approach:

1. by matching app names and package names
2. by matching component names in the manifest file, and
3. then by calculating the similarity between paths and apps.

We point out that a final manual verification is always performed to guarantee the correctness of the classification, which can also confirm the effectiveness of our heuristics. During this stage, we also collect one more piece of information: the code size of pre-loaded apps measured by their lines of code (LOC)[38]. Although it is impossible for us to get all the source code of the pre-loaded apps, we can still roughly estimate their size based on their decompiled .smali code. Therefore, we can draw a rough estimate of vendor customization from provenance analysis because the number and code size of apps are important indicators.

VII. PERMISION USAGE ANALYSIS

Our next analysis stage is designed to detect instances of permission overprivilege, where an app requests more permissions than it uses. ASEF applies permission usage analysis to measure the adoption of the principle of least privilege in app development. Note that here it is only possible to get the usage of permissions defined in the standard AOSP framework. The usage of vendor-specific permissions cannot be counted because of the lack of related information. There are four types of permissions in Android:

- normal
- dangerous
- system
- system Or Signature.

The latter three are sensitive, because normal permissions are not supposed to be privileged enough to cause damage to the user. Specifically, we define permissions declared by element uses-permission in the manifest file as requested permissions, and permissions which are actually used (e.g., by using related APIs[39]) as used permissions respectively. An over declared permission is a permission which is requested but not used. Over privileged apps contain at least one over declared permission. In permission usage analysis, from the database, we have the initial requested permission set of apps (as it is in the manifest information), and our goal is to find the over declared permission set. Despite the initial requested permission set, we find it still needs to be augmented. Especially, there is a special manifest file attribute, shared UserId, which causes multiple apps signed by the same developer certificate to share a user identifier, thus sharing their requested permission sets. (As permissions are technically assigned to a user identifier, not an app, all such apps will be granted the union of all the permissions requested by each app. Accordingly, apps with the same shared UserId require extra handling to get the complete requested permission set. Next, we leverage the known permission mapping built by earlier work to determine which permissions are actually used. Having built both the requested permission set and the used permission set, we can then calculate the over declared permission set.

Our approach to calculate the over declared permission set is conservative. Notice that some permissions declared in the manifest file may be deprecated in the corresponding standard Android framework. An example is the permission READ_OWNER_DATA that was removed after API level 8 (i.e., Android version 2.2), but still declared by one app in the Nexus 4 (API level 17, or Android 4.2). We do not consider them as over declared permissions, because the vendor may retain deprecated permissions in the customized framework for its own usage. Some studies concerned themselves only with those permissions that are available to third-party apps. In our study, we need to cover additional permissions defined at system and system Or Signature levels, which may not be well documented. After obtaining the over declared permission set, we then analyze the overall permission usage of each device, and classify results by provenance. The distributions of over privileged apps as well as over declared permissions can then both be studied. Further, we also perform horizontal and vertical analysis, i.e., cross-vendor, same generation, vs. cross-generation, same-vendor comparisons.

VIII. VULNERABILITY ANALYSIS

While our permission usage analysis aims to measure the software development practices used in the creation of each pre-loaded app, vulnerability analysis is concerned with finding real, actionable exploits within those apps. Specifically, we look for two representative types of vulnerabilities in the Android platform which stem from misunderstanding or misusing Android's permission system. First, we identify permission re-delegation attacks, which are a form of the classic confused deputy attack. Such an attack exists if an app can gain access to an Android permission without actually requesting it. A typical example is an app which is able to send Short Message Service (SMS) messages without acquiring the (supposedly required) SEND_SMS permission. For the second kind of vulnerability, we consider content leaks, which essentially combine the two types of content provider vulnerabilities: passive content leaks and content pollution. An unprotected content provider (i.e., one that takes no sensitive permission to protect its access) is considered to have a passive content leak if it is world-readable, and to have content pollution if it is world-writable. We extend this definition to cover both open and protected content providers. The protected ones are also interesting as there may also exist unauthorized accesses to them through the other three types of components which could serve as springboards for exploitation. For ease of presentation, we call these vulnerabilities content leaks. As our main goal is to accurately locate possible vulnerabilities, we in this study consider the following adversary model: a malicious app, which is compatible with the phone, may be installed on the phone by the user. We do not expect the malicious app will request any sensitive permission during installation, which means it, will only rely on vulnerable apps to accomplish its goals: either steal money from the user, gather confidential data, or maliciously destroy data. In other words, we limit the attacker to only unprivileged third-party apps to launch their attacks. Keeping this adversary model in mind, we focus our analysis on security-critical permissions – the ones that protect the functions that our adversary would most like to gain access to. Specifically, for permission re-delegation attacks, we focus on permissions that are able to perform dangerous actions, such as SEND_SMS and MASTER_CLEAR, because they may lead to serious damage to the user, either financially or in terms of data loss. As for content leaks, we ignore those whose exposures are likely to be intentional. Note that some apps may be vulnerable to low-severity content leaks; for example, publicly-available information about a network TV schedule is not as sensitive as the user's banking credentials. In other words, we primarily consider serious content leaks whose exposures are likely to cause critical damages to the user. To actually find these vulnerabilities, we rely on a few key techniques. An essential one is reachability analysis, which is used to determine all feasible paths from the entry point set of all Android components, regardless of whether we consider them to be protected by a sensitive permission. To better facilitate vulnerability analysis[41], we define two varieties of sinks:

- sensitive-sinks: sensitive Android APIs which are related to sensitive permissions (e.g., MASTER_CLEAR) of our concern
- bridge-sinks: invocations that are able to indirectly trigger another (vulnerable) component, e.g., sendBroadcast.

Note that any path reachable from an open entrypoint or component can be examined directly to see if it has a sensitive-sink. Meanwhile, we also determine whether it could reach any bridgelink that will trigger other protected components (or paths). The remaining paths, whose entry points are protected, are correlated with paths that contain bridge-sinks to form the complete vulnerable path, which is likely cross-component or even cross different apps. This is essentially a reflection-based attack. All calculated (vulnerable) paths will subject to manual verification. We stress that unlike some previous works which mainly focus on discovery of vulnerabilities; this analysis stage primarily involves a more contextual evaluation of vulnerabilities, including distribution, evolution and the impact of customization. Especially, we use the distribution of vulnerable apps as a metric to assess possible security impact from vendor customizations. Note the detected vulnerabilities are classified into different categories by their provenance and leveraged to understand the corresponding impactof customization. As mentioned earlier, both horizontal and vertical impact analyses are performed.

IX. REACHABILITY ANALYSIS

It is performed in two steps. The first step is intra-procedural reachability analysis, which involves building related call graphs and resolving it by conventional def-use analysis. The resolution starts from the initial state (pre-computed when the database is initially populated) and then gradually seeks a fixed point of state changes with iteration (due to various transfer functions). However, as the state space might be huge (due to combinatorial explosion), the convergence progress could be slow or even unavailable. In practice, we have to impose additional conditional constraints to control the state-changing iteration procedure. We call the result of intra-procedural analysis, i.e., the states of variables and fields, a summary. The second step is inter-procedural reachability analysis that is used to propagate states between different methods. After each propagation, method summaries might be changed. In such cases, intra-procedural reachability analysis is performed again on each affected method to generate a new summary. Inter-procedural reachability analysis is also an iterative process, but takes longer and requires more space to converge; therefore, we use some heuristics to reduce the computational and space overhead. For instance, if a variable or field we are concerned with has already reached a sink, there is no need to wait for convergence. Paths of apps from different vendors but with similar functionality may share something in common, especially for those apps inherited from the standard AOSP framework. Here “common” does not mean that their source code is exactly the same, but is similar from the perspective of structure and functionality. Many devices reuse the code from the AOSP directly, without many modifications. If we have already performed reachability analysis on such a common path, there is no need to do it on its similar counterparts. It improves system performance since reachability analysis is time consuming (especially when the state space is huge). Therefore, we also perform a similarity analysis as a part of the reachability analysis to avoid repetitive efforts.

X. REFLECTION ANALYSIS

To facilitate our analysis, we classify vulnerable paths into the following three types:

- in-component: a vulnerable path that starts from an unprotected component to a sink that is located in the same component.
- cross-component: a vulnerable path that starts from an unprotected component, goes through into other components within the same app, and then reaches a sink.
- cross-app: a vulnerable path that starts from an unprotected component of one app, goes through into another app’s components, and eventually reaches a sink.

The in-component vulnerable paths are relatively common and have been the subject of recent studies. However, the latter two, especially the cross-app ones, have not been well studied yet, which is thus the main focus of our reflection analysis. Note that a reflection-based attack typically involves with multiple components that may not reside in the same app. Traditional reachability analysis has been effective in detecting in-component vulnerable paths. However, it is rather limited for other cross-component or cross-app vulnerable paths. (A cross-app execution path will pass through a chain of related apps to ultimately launch an attack.) In order to identify them, a comprehensive analysis of possible “connection” between apps is necessary. To achieve that, our approach identifies not only possible reachable paths within each component, but also the invocation relationship for all components. The invocation relationship is essentially indicated by sending intent from one component to another, explicitly or implicitly. An explicit intent specifies the target component to receive it and is straightforward to handle. An implicit intent, on the other hand, may be sent anonymously without specifying the receiving component, thus requiring extra handling (i.e., intent resolution in Android) to determine the best one from the available components. In our system, ASEF essentially mimics the Android intent resolution mechanism by matching an intent against all possible <intent-filter> manifest declarations in the installed apps. However, due to the offline nature of our system, we have limited available information about how the framework behaves at run-time. Therefore, we develop the following two heuristics:

- A component from the same app is preferable to components from other apps.
- A component from a different app which shares the same sharedUserId is preferable to components from other apps.

XI. RESULTS

We have implemented ASEF (Android Security Evaluation Framework),a mix of Java code and Python scripts. This category contains apps that exist in the AOSP and may (or may not) be customized by the vendor. In our evaluation, we examined six representativephones (Table 1) released after 2012 by six popular vendors: Google, Samsung, HTC, LG, Motorola and Sony. The selected phone model either has great impact and is representative, or has huge market share. For example, Google’s phones are designed to be reference models for their whole generation; Samsung is the market leader which occupies 39.6%of smartphone market share in 2013. To analyze these phone images, our system requires on average 80 minutes to process each image (i.e., around 40 seconds per app) and then reports vulnerable paths for us to manually verify. Considering the off-line nature of our tool, we consider this acceptable, even though our tool could be optimized further to speed up our analysis.

Table 1. Six android devices we have used.

DEVICE	RELEASE DATE	ANDROID VERSION	BUILD NUMBER
Motorola Moto G4	November 2013	4.4.2	KXB20.25
Samsung Note 5	September 2012	4.3	JSS15J
Sony Xperia M	June 2013	4.3	15.4.A.0.23
LG Optimus L5	April 2013	4.1.2	JZ054K
Google Nexus 6P	October 2013	4.4.2	KOT49H
HTC One V	April 2012	4.0.3	CL98360

XII. PROVENANCE ANALYSIS

As mentioned earlier, the provenance analysis collects a wide variety of information about each device and classifies pre-loaded apps into three different categories. In Table 2, we summarize our results. Overall, these six devices had 546 pre-loaded apps, totaling 3,67,34,925 lines of code (in terms of decompiled .smali code). A further break-down of these apps show that, among these devices, there are on average 18.67 (20.5%), 57 (62.63%), and 15.33 (16.84%) apps from the categories of AOSP, vendor and thirdparty, respectively. Note that the apps in the AOSP category may so be customized or extended by vendors. As a result, the figures in the AOSP column of Table 2 should be considered an upper bound for the proportion of code drawn from the AOSP. Accordingly, on average, vendor customizations account for more than 83.15% of apps (or 78.34% of LOC[38]) on these devices. In our study, we selected one phone model for each vendor,from the current crop of Android 4.x phones. Table 1 shows the initial releasedate of each phone model, as reported by GSM Arena. As it turns out, these devices can be readily classified by their release dates.Some devices were released in the year 2012 and the rest were released in 2013. Such classification is helpful to lead to certainconclusions. For example, as one might expect, the complexity these devices is clearly increasing over time. In all cases, the 2013 products contain more apps and LOC than their 2012 counterparts. Specifically, the Samsung Note II has 87 apps with 81,54,224 LOC, and the HTC One V has 91 apps with 4660940 LOC. The number of apps and LOC increase 90.47% and 103.49%, respectively. The HTC One V had the most number of vendor specific apps (80.21%) followed by Sony Xperia M (74.78%), Motorola Moto G (62.8%), LG Optimus L5 (60.02%), Samsung Galaxy Note II (57.47%) and Google Nexus 5 (39%).Interesting fact is that vendor specific apps in Google Nexus 5 accounted only 39% of the total apps, least amongst the devices studied.

The stat is quite interesting and understandable because Nexus 5 is the flagship phone of Google which owns the Android OS and hence least vendor customization was expected in this phone. Our analysis also shows that, though the baseline AOSP is indeed getting more complicated over time – but vendor customizations are at least keeping pace with the AOSP. This trend is not difficult to understand, as vendors have every incentive to add more functionality to their newer products, especially in light of their competitors doing the same. The Google-branded phone (Nexus 5) is particularly interesting. It has relatively few apps, as it is designed to be reference design with only minor alterations to the core AOSP. The Nexus 5 has 100 apps counting to 71,70,961 LOC. The Nexus 5 is the 2ndmost complex – only the Samsung Galaxy Note II, which is well known for its extensive customization, has more LOC. We attribute this to the fact that the Nexus 5 includes newerversions of vendor-specific apps (e.g., Gmail, Maps,Youtube etc.) that have more functionality with larger code size. Meanwhile, we also observe these devices experience slow update cycles: the average interval between two updates for a phone model is about half a year! Samsung Galaxy Note II got an update after 14 months of its release, the highest timeframe for any phone in our study.Sony Xperia M got it in the least time (9 months). Overall, official updates can hardly be called timely, which thereby seriously affects the security of these devices.

Table 2. Provenance analysis of representative devices

Model	Total		AOSP apps		Vendor apps		3rd party apps	
	Apps	LOC	Apps	LOC	Apps	LOC	Apps	LOC
Motorola Moto G4	70	4091293	11	984365	44	2280396	15	826532
Samsung Note 5	87	8154224	14	2983630	50	2987621	23	2182973
Sony Xperia M	115	7544194	19	2168394	86	3926521	10	1459279
LG Optimus L5	83	5113313	22	959212	50	700966	11	1149935
Google Nexus 6P	100	7170961	34	1134446	39	3654761	27	2381754
HTC One V	91	4660940	12	783970	73	3391343	6	485627
Total	**546**	**36734925**	**112**	**9014017**	**342**	**16941608**	**92**	**8486100**

After determining the provenance of each pre-loaded app, our next analysis captures the instances of permission over-privilege, where an app requests more permissions than it uses. The results are as:

Table 3. Permission Usage Analysis

Model	Total	AOSP apps	Vendor apps	3rd party apps
Motorola Moto G4	85.18	8.41	61.08	15.59
Samsung Note 5	78.71	12.91	15.54	10.46
Sony Xperia M	82.98	10.86	59.55	12.57
LG Optimus L5	92.61	24.72	40.44	27.45
Google Nexus 6P	49.17	38.07	42.31	8.79
HTC One V	91.17	16.58	52.83	21.76
Avg.	**86.64**	**18.59**	**51.93**	**16.12**

On average, there is an alarming fact that across the six devices, 86.64% of apps are over-privileged. The most overprivileged are the vendor customized apps (51.93%) followed by AOSP apps (18.59%). The 3rd party apps are least over privileged and contribute only 16.12%. The LG Optimus L5 has the most overprivileged apps (92.61%) and Samsung Galaxy Note II has the least (78.71%). Even Google's reference device, Nexus 5, does not necessarily perform better than the others; the Nexus 5 has the third most overprivileged apps of all the devices. Note that the situation does not appearto be getting better as time goes on. The average percentage of overprivileged apps in the 2013 devices has decreased to 85.61%, compared to 87.96% of all apps on 2012 devices. The decrease is of a very lesser margin and is still hardly reassuring. Interestingly, our results show that the proportion of overprivileged pre-loaded apps is more than the corresponding result of third-party apps. The highest percentage of 3rd party overprivileged apps is in LG Optimus L5 (27.45%) and least is in Google Nexus 5 (8.79%). On the other hand, vendor customized overprivileged apps account for 61.08% in Motorola Moto G, the most, and 40.44% in LG Optimus L5, the least. The interesting fact is that the least value of vendor customized apps (40.44%) is nowhere close to the highest value of 3rd party apps (27.45%). We believe there are two main reasons:

1) Pre-loaded apps are more privileged than third-party apps, as they can request certain permissions not available to third-party ones.
2) Pre-loaded apps are more frequent in specifying the sharedUserId property, thereby gaining much (possibly unnecessary) permission set.

Specifically, if we take into account the provenance of each app, a different story emerges. Looking over the breakdowns in Table 3, both modest gains over time appear to be primarily attributable to a reduction in app over privilegeamong AOSP apps; vendors appear to be responsible for roughly the same amount of overprivileged apps in each image (57.82% of both 2012 apps, vs. 55.96% of all 2013 apps). In this sense the vendors themselves do notappear to care significantly more about the least privilege principlethan third-party app developers do, despite being, on average, muchlarger corporate entities.Note that pre-loaded apps have access to certain permissions that are not available to third-party apps. Therefore, these overdeclared permissions, if exploited, can lead to greater damage. For example, our results demonstrate that both REBOOT and MASTER_CLEAR are among overdeclared permissions that can allow for changing (important) device status or destroying user data without notification.

XIII. CONCLUSION

have come across as a user and researcher, but they could be subjective. There are certain features and enhancements which I would expect to be included in future versions, and new ones might be added as they become available or, rather invented.

My research covers the basics of Android, its functioning as a system and logging and debugging mechanism. In conclusion, Android's future is happening now, and further enhancements to this already diverse and this multifunctional platform will only take software engineering to the next level.

REFERENCES

[1] F. ABLESON, C. COLLINS, AND R. SEN, Unlocking Android, A Developer Guide, Manning Publications, Greenwich, Connecticut, 2009.

[2] OPEN HANDSET ALLIANCE, Mobile operators. Open handsetAlliance,http://www.openhandsetalliance.com/oha_members.html, accessed August 2012, n.d.

[3] A. HOOG, Android Forensics: Investigation, Analysis and Mobile Security for Google, Syngress, Waltham, Massachusetts, 2011.

[4] ANDROID BLOG, The Android boot process from power on. Android Blog,http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html, accessed October 2012, June 2009.

[5] C. GERIGAN AND P. OGRUTAN, AT Commands in Project based Learning, Bull.Transylvania University of Braşov Ser. I: Eng. Sciences, 4 (2011), pp. 115-122.

[6] TELIT COMMUNICATIONS, AT Commands Reference Guide, Telit Communications, Sgonico, Italy, 2012.

[7] ANDROID OPEN SOURCE PROJECT, Radio interface layer. Android Open Source Project, http://www.kandroid.org/onlin pdk/guide/telephony.html, accessed October 2012, n.d.

[8] TELECOM4U, Android: Radio Interface Layer (RIL). Telecom4u, http://telecom4u.in/home/androidradio-interface-layerril, accessed October 2012, n.d.

[9] ANDROID DEVELOPERS, Using DDMS. Android Developers,http://developer.android.com/tools/debugging/ddms.html, accessed September 2012, n.d.

[10] ANDROID DEVELOPERS, Android debug bridge. Android Developers, http://developer.android.com/tools/help/adb.html accessed August 2012, n.d.

[11] ANDROID DEVELOPERS, Reading and writing the logs. Android Developers, http://developer.android.com/tools/debugging/debugging-log.html, accessed October 2012, n.d.