The Impact of Vendor Customizations on Android Security

Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, Xuxian Jiang
Department of Computer Science
North Carolina State University
{lwu4, mcgrace, yajin_zhou, cwu10}@ncsu.edu, jiang@cs.ncsu.edu

ABSTRACT

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.

In this paper, we analyze ten representative stock Android images from five popular smartphone vendors (with two models from each vendor). 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* 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 85.78% of all pre-loaded 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 (except for Sony) arose from vendor customizations. In general, this pattern held over time – newer smartphones, we found, are not necessarily more secure than older ones.

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516728.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls; D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*

Keywords

Android; Provenance; Customization; Static Analysis

1. INTRODUCTION

The smartphone market has grown explosively in recent years, as more and more consumers are attracted to the sensor-studded multipurpose devices. According to IDC [30], smartphone vendors shipped a total of 482.5 million mobile phones in the fourth quarter of 2012 – levels nearly equal to those of feature phones. Meanwhile, among smartphones, Google's Android captured almost 70% of the global smartphone market share last year [31], compared to about 50% the year before [20].

Android's popularity is due in part to it being an open platform. Google produces a baseline version of Android, 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 work [24] has anecdotally shown that Android devices had security flaws shipped in their pre-loaded apps. Note that stock images include code from potentially many sources: the AOSP 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.

In this paper, we aim to study vendor customizations on stock Android devices and assess the impact on overall Android security. Especially, we intend to determine the source of the security issues that trouble Android smartphone images, then further determine how the situation is evolving over time. To that end, we develop 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: apps originating from the AOSP, apps customized or written by the vendor, and 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 overprivilege, 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.

To facilitate our analysis, we implement a Security Evaluation Framework for Android called SEFA to evaluate stock smartphone images. Given a particular phone firmware image, SEFA first preprocesses it and imports into a local database a variety of information about the image, including the number of apps and numerous information about each app, such as the list of requested permissions, declared components, and the set of used Android APIs. Then SEFA compares each pre-loaded app with various ones in the original AOSP to determine its source and further performs a system-wide data-flow analysis to detect possible vulnerabilities. In our study, we have applied SEFA to ten flagship phone models from five popular vendors: Google, Samsung, HTC, LG, and Sony. For each vendor, we selected two phones: one from the current crop of Android 4.x phones, and one from the previous generation of 2.x devices. This slate of devices allows us to do two comparative analyses: *horizontal* differential analysis compares the various manufacturers' offerings for a given generation, while *vertical* differential analysis studies the evolution of any given vendor's security practices chronologically.

Our evaluation results show that more than 81.78% of pre-loaded apps (or 76.34% of LOC) on stock Android devices are due to vendor customizations. It is worrisome to notice that vendor customizations were, on the whole, responsible for the bulk of the security problems suffered by each device. On average, 85.78% of all pre-loaded apps in examined stock images are overprivileged with a majority of them directly from vendor customizations. And vendor apps consistently exhibited permission overprivilege, regardless of generation. Our results also show that vendor customizations are responsible for a large proportion of the vulnerabilities in each phone. For the Samsung, HTC, and LG phones, between 64.71% and 85.00% of the vulnerabilities were due to vendor customizations. This pattern was largely stable over time, with the notable exception of HTC, whose current offering is markedly more secure than the last-generation model we evaluated.

The rest of this paper is organized as follows. We present our methodology and system framework in Section 2, and describe implementation and evaluation results with case studies in Section 3. We then discuss for possible improvements in Section 4. Finally, we compare with related work and conclude this paper in Section 5 and Section 6 respectively.

2. DESIGN

The goal of this work 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.

In Figure 1, we summarize the overall architecture of the proposed SEFA system. Our system takes a stock phone image as

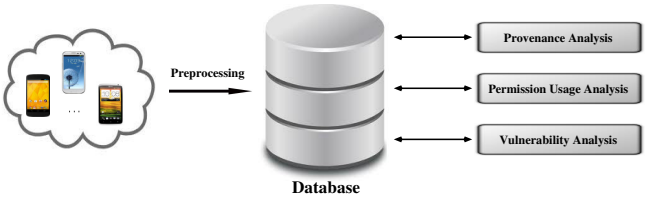


Figure 1: The overall architecture of SEFA

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 have developed three such analyses. First, to classify each app based on its presumed authorship, we perform *provenance analysis* (Section 2.1). 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. To further get a sense of the security and privacy problems posed by each app, we use two different analyses: *permission usage analysis* (Section 2.2) assesses whether an app requests more permissions than it uses, while *vulnerability analysis* (Section 2.3) 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.

2.1 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 pre-loaded 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 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 [29].

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 (Section 2.3) will use *paths*, sequences of methods that start at an entrypoint and flow into a *sink* (i.e., API 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 node (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; (3) and 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). 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.[1] Therefore, we can draw a rough estimate of vendor customization from provenance analysis because the number and code size of apps are important indicators. In addition, we also collect firmware release dates and update cycles as supplementary information for our later evaluation (Section 3.1).

2.2 Permission Usage Analysis

Our next analysis stage is designed to detect instances of permission overprivilege, where an app requests more permissions than it uses. SEFA 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* and *systemOrSignature*. 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) as *used permissions* respectively. An *overdeclared permission* is a permission which is requested but not used. *Overprivileged apps* contain at least one overdeclared permission.

Algorithm 1 outlines our process for 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 overdeclared permission set. Despite the initial requested permission set, we find it still needs to be augmented. Especially, there is a special manifest file attribute, `sharedUserId`, 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 `sharedUserId` require extra handling to get the complete requested permission set.) Next, we leverage the known permission mapping built by earlier work [2] to determine which permissions are actually used.[2] Having built both the requested permission set and the used permission set, we can then calculate the overdeclared permission set.

Algorithm 1: Permission Usage Analysis

Input: initial info. in our database about pre-loaded apps
Output: overdeclared permission set of apps

$apps$ = all apps in one image;
$mappings$ = all permission mappings;
A = API invocation set;
I = intent usage set;
C = content provider usage set;
S = shared user id set;
R = requested-permission set;
$U = \emptyset$; `// used permission set`
$O = \emptyset$; `// overdeclared permission set`

foreach $s \in S$ **do**
  $tmp = \emptyset$;
  foreach $app \in s$ **do**
    $tmp = tmp \cup R[app]$;
  foreach $app \in s$ **do**
    $R[app] = tmp$;

foreach $app \in apps$ **do**
  foreach $a \in A[app]$ **do**
    $U[app]$.add($mappings[a]$);
  foreach $i \in I[app]$ **do**
    $U[app]$.add($mappings[i]$);
  foreach $c \in C[app]$ **do**
    $U[app]$.add($mappings[c]$);

foreach $app \in apps$ **do**
  $O[app] = R[app] - U[app]$;
return O

Our approach to calculate the overdeclared 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`[3] 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 overdeclared permissions, because the vendor may retain deprecated permissions in the customized framework for its own usage.

[1] We use the *baksmali* [42] tool to translate the Android app's `.dex` code into the `.smali` format.

[2] Early studies including PScout [2] concern 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 *systemOrSignature* levels, which may not be well documented.

[3] Without specification, the prefix of standard Android permission name `android.permission.` is omitted in this paper.

After obtaining the overdeclared permission set, we then analyze the overall permission usage of each device, and classify results by provenance. The distributions of overprivileged apps as well as overdeclared 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.

2.3 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* [18], which are a form of the classic confused deputy attack [27]. 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 combines the two types of content provider vulnerabilities reported by Zhou and Jiang [52]: *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 permissions 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[4]. 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 entrypoint set of all Android components, regardless of whether we consider them to be protected by a sensitive permission (Section 2.3.1). To better facilitate vulnerability analysis, 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 bridge-link that will trigger other protected components (or paths). The remaining paths, whose entrypoints 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 and we will describe it in Section 2.3.2 in greater detail. All calculated (vulnerable) paths will subject to manual verification.

We stress that unlike some previous works (e.g., [24]) 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 impact of customization. As mentioned earlier, both horizontal and vertical impact analyses are performed.

2.3.1 Reachability Analysis

Our reachability analysis 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 [11]. 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. A more formal description of our reachability analysis is listed in Algorithm 3 (see Appendix A).

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. We believe this 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.

[4]Some content providers are exported explicitly, such as `TelephonyProvider` in the app of the same name.

2.3.2 *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 [24, 34]. 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. A concrete example that is detected by our tool will be shown in Figure 6 (Section 3.3.1).

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 an *intent* [21] 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, SEFA 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.

If multiple component candidates still exist for a particular intent, we simply iterate each one to report possible vulnerable path and then manually verify it in a real phone setting. In Algorithm 2, we summarize the overall procedure. The basic idea here is to maintain a visited component list. For a particular component, the algorithm returns $\emptyset$ if it has been visited; otherwise we add it into the list, and check all possible components that are able to start up that component recursively.

Algorithm 2: Reflection Analysis

```
Input: current component, visited component list
Output: vulnerable path set
/* find vulnerable paths recursively from
   backward for current component                */
c = current component;
VC = visited component list;
CC = components which are able to start up c;
V = ∅; // vulnerable path set

if c ∈ VC then
    return V
else
    VC.append(c);

foreach cc ∈ CC do
    tmp = VC.clone();
    V.add(Reflection Analysis(cc, tmp));
return V
```

3. IMPLEMENTATION AND EVALUATION

We have implemented a SEFA prototype as a mix of Java code and Python scripts with 11,447 and 4,876 lines of code (LOC) respectively. In our evaluation, we examined ten representative phones (Table 1) released between the end of 2010 and the end of 2012 by five popular vendors: Google, Samsung, HTC, LG, 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 2012 [30]. To analyze these phone images, our system requires on average 70 minutes to process each image (i.e., around 30 seconds per app) and reports about 300 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.

3.1 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 1, we summarize our results. Overall, these ten devices had 1,548 pre-loaded apps, totalling 114,427,232 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 28.2 (18.22%), 99.7 (64.41%), and 26.9 (17.38%) apps from the categories of AOSP, vendor and third-party, respectively. Note that the apps in the AOSP category may also be customized or extended by vendors (Section 2.1). As a result, the figures in the AOSP column of Table 1 should be considered an upper bound for the proportion of code drawn from the AOSP. Accordingly, on average, vendor customizations account for more than 81.78% of apps (or 76.34% of LOC) on these devices.

In our study, we selected two phone models for each vendor, one from the current crop of Android 4.x phones, and one from the previous generation of 2.x devices. Table 2 shows the initial release date of each phone model, as reported by GSM Arena [26]. As it turns out, these devices can be readily classified by their release dates: the past generation of devices all were initially released before 2012, while the current generation's products were released in 2012 or later. Therefore, we break them down into *pre-2012* and *post-2012* devices. Such classification is helpful to lead to certain conclusions. For example, as one might expect, the complexity of these devices is clearly increasing over time. In all cases, the post-2012 products from any given manufacturer contain more apps and LOC than their pre-2012 counterparts. Specifically, the HTC Wildfire S has 147 apps with 9,643,448 LOC, and the HTC One X has

[5]This category contains apps that exist in the AOSP and may (or may not) be customized by the vendor – Section 2.1. This definition also applies to Tables 3, 4 and 5.

Table 1: Provenance analysis of representative devices

			Total		AOSP app [5]		vendor app		third-party app	
Vendor	**Device**	**Version & Build #**	**# Apps**	**# LOC**	**# Apps (%)**	**# LOC (%)**	**# Apps**	**# LOC**	**# Apps**	**# LOC**
Samsung	Galaxy S2	2.3.4; I9100XWKI4	172	10,052,891	26 (15.12%)	2,419,155 (24.06%)	114	3,519,955	32	4,113,781
Samsung	Galaxy S3	4.0.4; I9300UBALF5	185	17,339,442	30 (16.22%)	6,344,721 (36.59%)	119	5,660,569	36	5,334,152
HTC	Wildfire S	2.3.5; CL362953	147	9,643,448	24 (16.33%)	2,759,415 (28.61%)	94	3,514,921	29	3,369,112
HTC	One X	4.0.4; CL100532	280	19,623,805	29 (10.36%)	4,718,633 (24.05%)	190	7,354,468	61	7,550,704
LG	Optimus P350	2.2; FRG83	100	6,160,168	27 (27.00%)	1,152,885 (18.72%)	40	604,197	33	4,403,086
LG	Optimus P880	4.0.3; IML74K	115	12,129,841	28 (24.35%)	3,170,950 (26.14%)	63	3,269,936	24	5,688,955
Sony	Xperia Arc S	2.3.4; 4.0.2.A.0.62	176	7,689,131	28 (15.91%)	1,164,691 (15.15%)	123	2,666,397	25	3,858,043
Sony	Xperia SL	4.0.4; 6.1.A.2.45	209	10,704,797	28 (13.40%)	1,800,690 (16.82%)	156	4,127,343	25	4,776,764
Google	Nexus S	2.3.6; GRK39F	73	5,234,802	31 (42.47%)	1,036,858 (19.81%)	41	2,821,874	1	1,376,070
Google	Nexus 4	4.2; JOP40C	91	15,848,907	31 (34.07%)	2,506,778 (15.82%)	57	12,156,673	3	1,185,456
Total			1548	114,427,232	282 (18.22%)	27,074,776 (23.66%)	997	45,696,333	269	41,656,123

Table 2: Release dates of examined Android devices

Device	**Release Date** [1]	**Update Date** [2] **& Version**
Nexus S	Dec 2010	4.0.4, Mar 2012; 4.1.2, Oct 2012
Nexus 4	Nov 2012	N/A
Wildfire S	May 2011	N/A
One X	May 2012	4.1.1, Nov 2012; 4.1.2, Mar 2013
Optimus P350	Feb 2011	N/A
Optimus P880	Jun 2012	4.1.2, Mar 2013
Galaxy S2	Apr 2011	2.3.6, Dec 2011; 4.0.3 Jul 2012; 4.1.1, Jan 2013; 4.1.2, Apr 2013
Galaxy S3	May 2012	4.1.1, Nov 2012; 4.1.2, Mar 2013
Xperia Arc S	Sep 2011	4.0.4, Aug 2012
Xperia SL	Sep 2012	4.1.2, May 2013

[1] The release dates may vary by area and carrier, but the difference is not that significant (e.g., around one month).

[2] The update dates collected here are mainly for markets in United States.

280 apps with 19,623,805 LOC. The number of apps and LOC increase 90.47% and 103.49%, respectively.

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 phones are particularly interesting. Both have relatively few apps, as they are designed to be reference designs with only minor alterations to the core AOSP. However, the Nexus 4 has over three times as many lines of code as were included in the Nexus S, despite adding only 18 apps. The Nexus S was the simplest phone we studied, while the Nexus 4 is the third most complex – only the HTC One X and Samsung Galaxy S3, which are well known for their extensive customization, have more LOC. We attribute this to the fact that the Nexus 4 includes newer versions of vendor-specific apps (e.g., Gmail) 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! Also, it is interesting to note that the updates in the United States tend to lag behind some other areas. For instance, users of the Samsung Galaxy S3 in the United Kingdom received updates to Android 4.1.1 (which was released in July 2012) in September 2012, while the updates were not available for users in the United States until January 2013. If we compare the update dates in Table 1 with the corresponding dates of Android releases [22], it often takes half a year for vendors to provide an official update (excepting Google's reference phones) for users in the United States, though in many cases carriers may share the blame. Overall, official updates can hardly be called timely, which thereby seriously affects the security of these devices.

3.2 Permission Usage Analysis

After determining the provenance of each pre-loaded app, our next analysis captures the instances of permission overprivilege, where an app requests more permissions than it uses. The results are summarized in Table 3. On average, there is an alarming fact that across the ten devices, 85.78% of apps are overprivileged. Even Google's reference devices do not necessarily perform better than the others; the Nexus S has the second most overprivileged apps of all the pre-2012 devices. Note that the situation appears to be getting slightly better as time goes on. The average percentage of overprivileged apps in the post-2012 devices has decreased to 83.61%, compared to 87.96% of all apps on pre-2012 devices. But this situation is still hardly reassuring. Interestingly, our results show that the proportion of overprivileged pre-loaded apps are more than the corresponding result of third-party apps (reported by [2, 17]). 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 many (possibly unnecessary) permissions.

Specifically, if we take into account the provenance of each app, a different story emerges. Looking over the breakdowns in Table 3, the modest gains over time appear to be primarily attributable to a reduction in app overprivilege among AOSP apps; vendors appear to be responsible for roughly the same amount of overprivileged apps in each image (51.29% of all pre-2012 apps, vs. 52.71% of post-2012 apps). In this sense the vendors themselves do not appear to care significantly more about the least privilege principle than third-party app developers do, despite being, on average, much larger corporate entities.

In Figure 2, we summarize the distributions of overprivileged apps among pre-2012 devices and post-2012 devices respectively. The majority of overprivileged apps have no more than 10 overdeclared permissions. 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.

3.3 Vulnerability Analysis

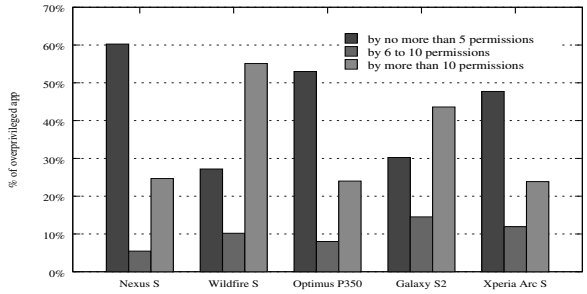
Our vulnerability analysis led to several interesting results, especially once combined with our efforts to determine the provenance of each app. In particular, if we consider the distribution of vulnerable apps across each phone image (Table 4), the percentage of vulnerable apps of these devices varies from 1.79% to 14.97%. Applying our horizontal analysis to each generation of devices, it appears that the HTC Wildfire S and LG Optimus P880 have the

Table 3: Permission usage analysis of representative devices

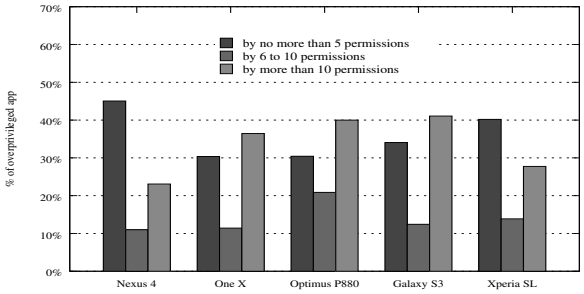
Pre-2012 devices					Post-2012 devices				
Device	% of overprivileged apps among all pre-loaded apps				Device	% of overprivileged apps among all pre-loaded apps			
	Total	AOSP app [5]	vendor app	third-party app		Total	AOSP app [5]	vendor app	third-party app
Nexus S	90.41%	38.36%	50.68%	1.37%	Nexus 4	79.12%	30.77%	48.35%	0.00%
Wildfire S	92.52%	15.65%	57.82%	19.05%	One X	78.21%	10.00%	50.71%	17.50%
Optimus P350	85.00%	23.00%	33.00%	29.00%	Optimus P880	91.30%	20.87%	50.43%	20.00%
Galaxy S2	88.37%	13.95%	58.14%	16.28%	Galaxy S3	87.57%	15.14%	55.68%	16.76%
Xperia Arc S	83.52%	14.20%	56.82%	12.50%	Xperia SL	81.82%	11.96%	58.37%	11.48%
Average	**87.96%**	**21.03%**	**51.29%**	**15.64%**	**Average**	**83.61%**	**17.75%**	**52.71%**	**13.15%**
Overall Average: 85.78%									

Table 4: Vulnerability analysis of representative devices (I)

Pre-2012 devices					Post-2012 devices				
Device	% of vulnerable apps among all apps				Device	% of vulnerable apps among all apps			
	Total	AOSP app [5]	vendor app	third-party app		Total	AOSP app [5]	vendor app	third-party app
Nexus S	5.48%	2.74%	2.74%	0.00%	Nexus 4	2.20%	1.10%	1.10%	0.00%
Wildfire S	14.97%	4.76%	8.84%	1.36%	One X	1.79%	0.71%	0.71%	0.36%
Optimus P350	11.00%	4.00%	1.00%	6.00%	Optimus P880	10.43%	5.22%	5.22%	0.00%
Galaxy S2	12.21%	3.49%	6.98%	1.74%	Galaxy S3	6.49%	2.70%	3.24%	0.54%
Xperia Arc S	2.27%	1.14%	0.00%	1.14%	Xperia SL	1.91%	0.96%	0.48%	0.48%
Average	**8.99%**	**3.23%**	**3.91%**	**1.85%**	**Average**	**4.56%**	**2.14%**	**2.15%**	**0.28%**
Overall Average: 6.77%									



(a) Pre-2012 devices



(b) Post-2012 devices

Figure 2: Distributions of overprivileged apps

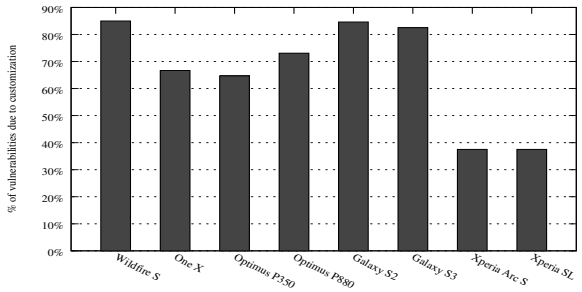


Figure 3: Vulnerabilities due to vendor customizations

most vulnerable apps in each generation. Inversely, the Sony Xperia Arc S and – interestingly – the HTC One X have the least vulnerable apps by proportion. As one may expect, Google's reference phones (especially the Nexus 4) both perform well compared to their contemporaries, as their images are designed to be a reference baseline. Our vertical analysis has an even more impressive result – the percentage of vulnerable apps across each generation dropped from an average of 8.99% last generation to 4.56% this generation. Even incorporating provenance information in the vertical analysis, there is a dramatic improvement for all three categories of app.

However, the percentage of vulnerable apps is not necessarily a good metric to measure the security of such devices. The complexity metrics we collected as part of our provenance analysis (see Table 1) show that the devices contain ever-increasing amounts of code. Therefore, while fewer vulnerabilities – as a percentage – may be introduced in newer phones, the sheer scale of their stock images works to counteract any gains. Furthermore, some vulnerable apps may contain more vulnerabilities than others. As a result, there is value in counting the absolute number of critical vulnerabilities as well as the proportion of vulnerable apps.

To this end, we summarize our results in that format in Table 5. When we use this table for vertical differential analysis, it tells quite a different story. While, indeed, the number of vulnerabilities in each generation of devices generally decreased, the reduction is nowhere near so dramatic. Furthermore, when considered in horizontal differential analysis, different devices take the crown for most and least secure devices in each generation. The HTC Wildfire S is still the least secure pre-2012 device, but only by a hair – the Samsung Galaxy S2 has only one fewer vulnerability. The Sony Xperia Arc S is tied with the Google Nexus S for the most secure pre-2012 device. Meanwhile, there is a complete shake-up among the post-2012 devices: the Samsung Galaxy S3 has 40 vulnerabilities to the LG Optimus P880's 26, while the HTC One X (at 15 vulnerabilities) falls to mid-pack, behind the Nexus 4 (at 3) and the Sony Xperia SL (at 8).

Table 5 still does not tell the complete story. Looking at the table's provenance analysis results, it appears that most of the vulnerabilities stem from the AOSP. However, recall that our provenance analysis concerns the original provenance of each *app*, not each *vulnerability*. To gather information about the provenance of each vulnerability, we manually examine each of the reported vulnerable paths. Our results are shown in Figure 3 (note that Google's phones are not included here because the AOSP is led by Google, making the distinction difficult). For the Samsung, HTC, and LG phones, the majority of vulnerabilities – between 64.71% and 85.00% – did *not* originate from the AOSP. However, for both of Sony's products,

Table 5: Vulnerability analysis of representative devices (II)

Pre-2012 devices					Post-2012 devices				
Device	# of vulnerabilities				Device	# of vulnerabilities			
	Total	AOSP app [1]	vendor app	third-party app		Total	AOSP app [1]	vendor app	third-party app
Nexus S	8	6	2	0	Nexus 4	3	2	1	0
Wildfire S	**40**	23	15	2	One X	15	12	2	1
Optimus P350	17	11	1	5	Optimus P880	26	17	9	0
Galaxy S2	**39**	18	18	3	Galaxy S3	**40**	20	12	8
Xperia Arc S	8	6	0	2	Xperia SL	8	6	1	1

[1] This category contains apps that exist in the AOSP and may (or may not) be customized by the vendor – Section 2.1.

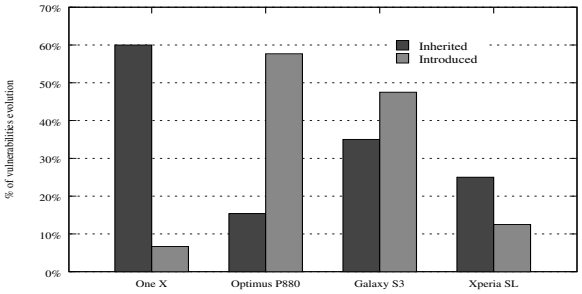


Figure 4: Vendor-specific vulnerability evolution

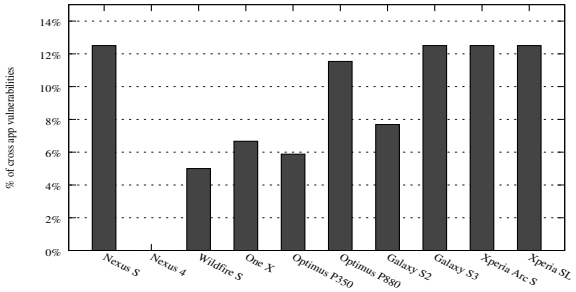


Figure 5: Distribution of cross-app vulnerabilities

only 37.50% of vulnerabilities were caused by vendor customizations. In fact, one of Sony's modifications to the AOSP actually mitigated a pre-existing bug in it.

We can also apply vertical differential analysis to this data, and therefore look at the evolution of vulnerabilities over time. The post-2012 devices may have *inherited* some vulnerabilities that were never caught during the lifetime of the pre-2012 devices, as they often have code in common with earlier devices by the same manufacturer[6]; alternatively, they may have *introduced* new vulnerabilities in their new and altered features. Figure 4 depicts this evolutionary information, which varies wildly in proportion between different vendors. For example, for the HTC One X, about 60.00% of its vulnerabilities were inherited from the previous device, while the Samsung Galaxy S3 has more introduced vulnerabilities than inherited ones (47.50% vs. 35.00%).

Table 6: Classification of detected vulnerabilities

Name	Description
CALL_PRIVILEGED	Initiate a phone call (including emergency number) without requiring confirmation
MASTER_CLEAR	Wipe out user data and factory reset
REBOOT	Reboot the device
RECORD_AUDIO	Allows an application to record audio.
SEND_SMS	Allows an application to send SMS messages.
SHUTDOWN	Power off the device
WRITE_SMS	Allows an application to write SMS messages
OTHER	All the other dangerous/critical operations

No treatment of this topic would be complete without discussing the distribution of vulnerabilities that we found. Table 6 lists the vulnerabilities we focus on. We use the names of permission to represent the most common (i.e., shared by devices of different vendors) vulnerabilities for permission re-delegation attacks with explicit permission names, and use `OTHER` to represent all other vulnerabilities (including both types of studied vulnerabilities). Note that vulnerabilities belong to `OTHER` do not mean they are not critical, and the only reason is that they are more vendor- and model-specific. Table 7 lists the distribution of these vulnerabilities.

With these detected vulnerabilities, we have attempted to contact the corresponding vendors. While some of them have already been confirmed, other vendors have still not spoken with us even after several months. Among the detected vulnerabilities, we believe cross-app ones are the most challenging to detect. Figure 5 gives the percentage of cross-app vulnerabilities. On average, 8.90% vulnerabilities are of this type. In the following, we describe some specific vulnerabilities introduced due to vendor customizations.

[6] The relationship between each pair of devices may be not direct (i.e., predecessor and successor), but we can still regard these vulnerabilities as an inheritance because they are vendor-specific.

3.3.1 *Case Study: Samsung Galaxy S3*

In the Samsung Galaxy S3, there is a pre-loaded app named *Keystring_misc*. This particular app has a protected component `PhoneUtilReceiver` that leads to a dangerous path for performing a factory reset, thus erasing all user data on the device. This path ends in the `phoneReset` method, which will broadcast an intent `android.intent.action.MASTER_CLEAR` to perform the operation. At first sight, this path seems to be *safe* because this component is protected by the `com.sec.android.app.phoneutil.permission.KEYSTRING` permission, which is defined with the restrictive *systemOrSignature* protection level (i.e., only other firmware apps, or apps from Samsung, can invoke it).

Unfortunately, there exists another app named *FactoryTest* that contains a feasible path which is able to start up this very component in the *Keystring_misc* app. This arrangement is an example of a cross-app vulnerable path, which can be used to launch a reflection attack (Figure 6). Specifically, this app exports a service called `FtClient` without any protection. After being launched, the service will start a thread and then try to build connections with two hard-coded local socket addresses: `FactoryClientRecv` and `FactoryClientSend`. The established connections can be exploited to send commands through the first socket. Our manual investigation shows that there are many dangerous operations that can be triggered by this exploit, including `MASTER_CLEAR`, `REBOOT`, `SHUTDOWN` and `SEND_SMS`.

In our study, we also discover a number of other vulnerabilities. For example, there are four content providers in the *sCloudBackupProvider* app (with the package name of `com.sec.android.sCloudBackupProvider`). They expose access interfaces to specific databases, including `calllogs.db`, `sms.db`, `mms.db` and `settings.db`. Each of them is protected by two permissions, but with normal (non-sensitive) protection levels. Apparently, they are accessible to any

Table 7: Distribution of vulnerabilities among examined devices

Name	# of vulnerabilities									
	Google		HTC		LG		Samsung		Sony	
	Nexus S	Nexus 4	Wildfire S	One X	P350	P880	Galaxy S2	Galaxy S3	Xperia Arc S	Xperia SL
CALL_PRIVILEGED	1	0	2	1	1	3	2	4	1	1
MASTER_CLEAR	0	0	1	0	0	0	3	2	0	0
REBOOT	0	0	0	0	0	1	4	4	0	0
RECORD_AUDIO	0	0	1	0	0	1	1	1	0	0
SEND_SMS	3	2	6	3	4	4	7	7	3	3
SHUTDOWN	0	0	0	0	0	0	0	1	0	0
WRITE_SMS	2	0	5	6	3	3	4	4	2	2
OTHER	2	1	25	5	9	14	18	17	2	2
Total	8	3	40	15	17	26	39	40	8	8

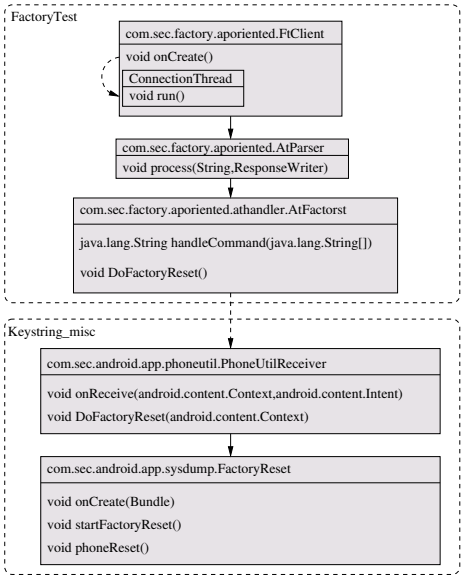


Figure 6: An example path of reflection attack

third-party app. Also, notice that this app exports four receivers without any protection, and it is able to craft specific intents that trigger corresponding backup actions to save various private information into these databases (e.g., standard SMS messages stored in `mmssms.db` will be copied into `sms.db`). After that, any app can simply retrieve these sensitive information (e.g., SMS messages and contacts information) through the corresponding four content providers without acquiring any sensitive permission.

3.3.2 Case Study: LG Optimus P880

The LG Optimus P880 has a number of vulnerabilities; here, we will detail two of them, leading to a permission re-delegation attack and a content leak respectively. However, unlike the Samsung vulnerabilities, neither of the ones we will describe are reflection attacks, making them easier to detect, describe, and exploit.

The first one is related to `REBOOT`, a permission reserved for system or pre-loaded apps. In the LG Optimus P880, there is an app named *LGSettings*, which is a customized version of the AOSP *Settings* app. This particular app exports an activity `com.android.settings.Reboot` (completely without permission protection). This activity will be triggered if an `android.intent.action.MAIN` intent is received, and its invocation will simply reboot the device directly. Note that the AOSP does not have the corresponding vulnerable component.

The second one is a rather direct content leak vulnerability. The `com.lge.providers.lgemail` content provider in the *LGEmail* app is not protected, and therefore exposes access to the *EMAIL.db*, a database that contains three tables named *EAccount*, *EMessageBox* and *EMessage*. These tables are very sensitive, as through them, all account and message related information (including message contents) can be exposed. Note that this app is customized from the AOSP *Email* app; however, in the AOSP, the corresponding content provider is protected by a permission named `com.android.email.permission.ACCESS_PROVIDER` with the *systemOrSignature* protection level. Therefore, LG's customization here adds a vulnerability to an otherwise-secure AOSP app.

4. DISCUSSION

While collecting the data for our evaluation, we saw some indirect evidence of software development policies in place at the various vendors. This evidence may be anecdotal, but we feel it is interesting enough to warrant mention. For example, Sony's standout performance does not appear to be accidental; in both their devices that we studied, the `eventstream` content provider (which was implemented as an SQLite database, as many are) actually had explicit checks for SQL injection attacks. Furthermore, Sony's customized version of the AOSP *Mms* app actually mitigated problems found in the unaltered problem. Similarly, as we remarked in Section 2.3, HTC made considerable progress between the release of the HTC Wildfire S and the One X, possibly due to early exposure of a large proportion of security vulnerabilities in earlier HTC's devices [39] and the efforts made by the corporation to take security to heart ever since. The One X makes extensive use of custom permissions to prevent further vulnerabilities from creeping into its firmware – a relatively straightforward approach to take, yet an effective one, as shown by our vulnerability analysis.

We also note that there are not very strong correlations between a number of superficial metrics and the number of vulnerabilities present in a stock firmware image. Code size does not strongly correlate: the Nexus 4 has the third-largest number of LOC, but the lowest number of vulnerabilities in the whole study. The number of apps does not correlate: both Sony devices perform very well, despite having a very large number of apps, while the LG devices do poorly on security even though they have the fewest apps of any non-reference device. Finally, even popularity does not appear to correlate with security: Samsung's Galaxy S3 was the most popular smartphone of 2012, having shipped in excess of 50 million units as of March 2013 [45], and yet it had the most vulnerabilities of any phone in its generation studied in this paper.

Lastly, we would like to acknowledge some of the limitations of our work. We do not cover the customization of system level code, which can be an interesting topic for future research. Our current prototype also has several constraints. First of all, our static analysis produces a relatively high false positive rate. On average, our analysis produces less than 300 paths per device. While it does not make too much effort to manually verify each path, it would be better if we could use a light-weight dynamic analyzer to reduce the

manual workload. Secondly, we generate the call graph recursively. In order to avoid very deep (potentially infinite) recursion, we constrain our analysis in two ways: only acyclic graphs are allowed, and/or the maximum length of a path (or the maximum exploration time) is set as an upper boundary for safety. These constraints may prevent us from discovering certain vulnerabilities if there exists heavy code obfuscation that either extends the length of vulnerable path or modifies the sinks we use to define such paths. Fortunately, it is our experience that most pre-loaded apps (other than bundled third-party apps) are not extensively obfuscated. As a result, these constraints were primarily triggered by infinite recursion, not by overly long, yet valid, call chains.

5. RELATED WORK

Provenance Analysis In our system, provenance provides important context which we use when evaluating the results of our other analyses. However, determining the provenance of an app or a piece of code is a difficult problem, which has attracted much research. For example, traditional similarity measurement has been widely used in malware (e.g., virus and worm) clustering and investigation. Kruegel et al. [32] propose an approach to detect polymorphic worms by identifying structural similarities between control flow graphs. SMIT [29], a malware database management system based on malware's function-call graphs, is able to perform large scale malware indexing and queries. Note these approaches are mainly developed for PC malware and have considerably more sophistication – and complexity – than our own. In the same spirit, we use chains of method signatures to determine whether a pre-loaded app is, in fact, an altered version of a core AOSP app. Previous insights however should be able to further guide us in improving accuracy and completeness of our approach.

In the Android platform, one line of inquiry concerns detecting repackaged apps – apps that have been altered and re-uploaded by some party other than the original author. DroidMOSS [50], DNADroid [9] and PiggyApp [49] all focus on detecting repackaged apps in Android app markets, while AppInk [46] pays attention to deterring repackaging behavior with watermarking techniques. These efforts all must establish the ancestry of any given app, which mirrors our efforts to understand, longitudinally, the evolution of firmware images using vertical differential analysis. However, we are not as concerned with the arrow of time – it is obvious the relationship between the original AOSP image and the vendor's customizations, for legal reasons more than technical ones.

Lastly, mobile advertisement libraries have attracted a lot of attention, as they live within other apps and share their permissions. They have been demonstrated to be another important source of privacy leaks [23]; furthermore, Book et al.'s longitudinal study [5] shows that negative behaviors may be growing more common in such libraries over time. As a result, several mitigation measures have been proposed. Some add new APIs and permissions [36] to attempt to isolate such alien code; AdSplit [41], in contrast, moves advertisement code into another process to allow the core Android framework to issue it different permissions, and thus enforce a different policy. Our work has a strange kinship with these works, in that we similarly are interested in poorly-tagged vendor customizations mixed in with code from other sources. While we operate on a different scale to evaluate whole phone images instead of individual third-party apps, many of the same concepts apply. Furthermore, there similarly exists a disconnection between the trust a user may afford the open-source, heavily vetted AOSP and the vendor's customizations of it – attempting to mitigate the flaws introduced by the vendor would be an interesting topic for future work.

Permission Usage Analysis Our permission usage analysis is built upon the accumulated insight of a number of other works in this area. For example, Stowaway [17], Vidas et al. [44] and PScout [2] all study the problem of overprivileged third party apps and provide permission mappings in different ways. Barrera et al. [3] study the permission usage patterns of third party apps by applying self-organizing maps. None of these works analyze the problem of permission overprivilege in pre-loaded firmware apps, which is one key focus of our work.

Others have attempted to infer certain security-related properties about apps based solely on their requested permissions. Kirin [16], for example, looks for hard-coded dangerous combinations of permissions to warn the user about potential malware. Sarma et al. [40], Peng et al. [37] and Chakradeo et al. [7], on the other hand, use machine learning techniques to automatically classify apps as potentially malicious based on the permissions they seek. In this initial study, we do not attempt to look for such emergent effects in the permissions requested by a pre-loaded app as the examined stock images are released by reputable and trustworthy entities.

Vulnerability Analysis Several works have attempted to survey the landscape of malware on Android (e.g., MalGenome [51]) as well as general apps [15, 38]. Other works, like DroidRanger [53], RiskRanker [25], Peng et al. [37] and MAST [7] all have been concerned with finding malicious apps in app markets that contain a large number of benign apps. DroidScope [48] uses virtualization to perform semantic view reconstruction, much like a number of desktop systems, to analyze Android malware. The insights developed in these works are useful in informing our own about the potential dangers of third-party malicious apps.

Permission re-delegation vulnerabilities, a form of the classic confused-deputy attack [27], have been known to be a problem on the Android platform for some time. ComDroid [8], Woodpecker [24], and CHEX [34] all apply static analysis techniques to find vulnerabilities in either third party or pre-loaded apps. SEFA is most similar to the latter two systems. But our system also performs provenance analysis, allowing us to determine the impact of vendor customizations on security. While ComDroid and Woodpecker only could detect in-component vulnerabilities, CHEX could additionally detect cross-component ones. Our work is the most comprehensive yet, as it can find in-component, cross-component, and cross-app vulnerabilities. Specifically, cross-app vulnerabilities account for 8.90% of the vulnerabilities that we found, a significant proportion that also leads to similar, if not greater, security risks.

Several systems [6, 12, 18] aim to mitigate the permission re-delegation problem by either checking IPC call chains or by monitoring the run-time communication between apps. Other works try to protect security in a different manner. For example, virtualization techniques are leveraged by Cells [1] and L4Android [33]. MoCFI [10] implements a control-flow integrity enforcement framework for apps on iOS platform. These works are all complementary to our own, as similar techniques may be able to be applied to mitigate the impact of the flaws that we have detected.

Several systems, including TaintDroid [14], PiOS [13], Apex [35], MockDroid [4], TISSA [54], AppFence [28], Aurasium [47], SORBET [19] and CleanOS [43] aim to study privacy leak issues. They all try to protect (or mitigate) the privacy leak problem by modifying the underlying framework. There has been considerably less work related to content leaks. ContentScope [52] tries to identify vulnerabilities related to third-party, unprotected content providers. Our own work uses a similar concept, but is concerned with content providers in firmware. Note these content providers are only ever found in that context and we have to additionally cover protected content providers (that have not been addressed by earlier work).

6. CONCLUSION

In this paper, we evaluate the security impact of vendor customizations on Android devices by designing and implementing the SEFA analysis framework. This tool performs several analyses to study the provenance, permission usage and vulnerability distribution of the pre-loaded apps that make up a device's firmware image. We evaluated ten devices from five different vendors: two models from each vendor, representing two different generations. We then compare the various vendors' offerings for a given generation, as well as the evolution of any given vendor's security practices over time. Our results show that due to heavy vendor customizations, on average, over half of the apps in each image are overprivileged vendor apps, and more than 60% of the vulnerabilities we identified stemmed from the vendors' modifications to the firmware. Furthermore, for most of the manufacturers in our study, these patterns were stable over time, highlighting the need for heightened focus on security by the smartphone industry.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. We also want to thank Kunal Patel, Wu Zhou and Minh Q. Tran for the helpful discussion. This work was supported in part by the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[2] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[3] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, 2010.

[4] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th International Workshop on Mobile Computing Systems and Applications*, HotMobile '11, 2011.

[5] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal Analysis of Android Ad Library Permissions. In *IEEE Mobile Security Technologies*, MoST '13, 2013.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS '12, 2012.

[7] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proceedings of the 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, 2013.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, 2011.

[9] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of 17th European Symposium on Research in Computer Security*, ESORICS '12, 2012.

[10] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS '12, 2012.

[11] Defuse. Use-define chain. `http://en.wikipedia.org/wiki/Use-define_chain`.

[12] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, 2011.

[13] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Symposium on Network and Distributed System Security*, NDSS '11, 2011.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, USENIX OSDI '10, 2010.

[15] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, 2011.

[16] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[17] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[18] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, 2011.

[19] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and Enhancing Android's Permission System. In *Proceedings of 17th European Symposium on Research in Computer Security*, ESORICS '12, 2012.

[20] Gartner. Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth. `http://www.gartner.com/it/page.jsp?id=1924314`.

[21] Google. Intent. `http://developer.android.com/reference/android/content/Intent.html`.

[22] Google. Platform Versions. `http://developer.android.com/about/dashboards/index.html`.

[23] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '12, 2012.

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

[25] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services*, MobiSys '12, 2012.

[26] Gsmarena. `http://www.gsmarena.com/`.

[27] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22, October 1988.

[28] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[29] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-Scale Malware Indexing Using Function-Call Graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[30] IDC. Strong Demand for Smartphones and Heated Vendor Competition Characterize the Worldwide Mobile Phone Market at the End of 2012, IDC Says. `https://www.idc.com/getdoc.jsp?containerId=prUS23916413#.UQIPbh0qaSp`.

[31] J. Koetsier. Android captured almost 70% global smartphone market share in 2012, Apple just under 20%. http://venturebeat.com/2013/01/28/android-captured-almost-70-global-smartphone-market-share-in-2012-apple-just-under-20/.

[32] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection*, RAID '05, 2005.

[33] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM '11, 2011.

[34] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[35] M. Nauman, S. Khan, and X. 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, 2010.

[36] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, 2012.

[37] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[38] S. Rosen, Z. Qian, and Z. M. Mao. AppProïňĄler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End Users. In *Proceedings of the 3nd ACM Conference on Data and Application Security and Privacy*, CODASPY '13, 2013.

[39] A. Russakovskii. http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phone-numbers-gps-sms-emails-addresses-much-more/.

[40] B. Sarma, C. Gates, N. Li, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android Permissions: A Perspective Combining Risks and Benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, 2012.

[41] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the 21th USENIX Security Symposium*, USENIX Security '12, 2012.

[42] Smali. An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/.

[43] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposure With Idle Eviction. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, USENIX OSDI '12, 2012.

[44] T. Vidas, N. Christin, and L. F. Cranor. Curbing Android permission creep. In *Proceedings of the 2011 Web 2.0 Security and Privacy Workshop*, W2SP '11, 2011.

[45] Wiki. Samsung Galaxy S3. http://en.wikipedia.org/wiki/Samsung_Galaxy_S_III.

[46] Z. Wu, X. Zhang, and X. Jiang. AppInk: Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '13, 2013.

[47] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21th USENIX Security Symposium*, USENIX Security '12, 2012.

[48] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21th USENIX Security Symposium*, USENIX Security '12, 2012.

[49] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, Scalable Detection of 'Piggybacked' Mobile Applications. In *Proceedings of the 3nd ACM Conference on Data and Application Security and Privacy*, CODASPY '13, 2013.

[50] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY '12, 2012.

[51] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, IEEE Oakland '12, 2012.

[52] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, NDSS '13, 2013.

[53] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS '12, 2012.

[54] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, 2011.

APPENDIX

A. REACHABILITY ANALYSIS

Algorithm 3: Reachability Analysis

Input: path from entrypoint to sink
Output: path is reachable or not

$ret = false$;
intra_analysis(all nodes in path) ;
$nodes$ = nodes in the path;
$edges$ = edges in the path;

while *constraint does not meet* **do**
  $flag = false$;
  foreach $n \in nodes$ **do**
    $callee$ = callee set of n;
    if $callee = \emptyset$ **then**
      break;
    foreach $c \in callee$ **do**
      if $(n, c) \in edges$ **then**
        $flag$ = $flag \cup c$.summarize(n);
  if $flag$ **then**
    inter_analysis(c);
    if *constraint meets* **then**
      break;
  else
    break;

ret = reachability_check($path$ with summary information);
return ret