Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis

Yousra Aafer, Xiao Zhang, and Wenliang Du
Syracuse University
{yaafer, xzhang35, wedu}@syr.edu

Abstract

Android customization offers substantially different experiences and rich functionalities to users. Every party in the customization chain, such as vendors and carriers, modify the OS and the pre-installed apps to tailor their devices for a variety of models, regions, and custom services. However, these modifications do not come at no cost. Several existing studies demonstrate that modifying security configurations during the customization brings in critical security vulnerabilities. Albeit these serious consequences, little has been done to systematically study how Android customization can lead to security problems, and how severe the situation is. In this work, we systematically identified security features that, if altered during the customization, can introduce potential risks. We conducted a large scale differential analysis on 591 custom images to detect inconsistent security features. Our results show that these discrepancies are indeed prevalent among our collected images. We have further identified several risky patterns that warrant further investigation. We have designed attacks on real devices and confirmed that these inconsistencies can indeed lead to actual security breaches.

1 Introduction

When vendors, such as Samsung, LG and HTC, put Android AOSP OS on their devices, they usually conduct extensive customization on the system. The reasons for customization can be many, including adding new functionalities, adding new system apps, tailoring the device for different models (e.g., phone or tablet), or carriers (e.g., T-mobile and AT&T), etc. Further complicating the process is Android updates pushed to the devices: the updates might target a new Android or app version.

This fragmented eco-system brings in several security risks when vendors change the functionalities and configurations without a comprehensive understanding of their implications. Previous work has demonstrated some aspects of these changes and the resulting risks. Wu et al. [25] analyze several stock Android images from different vendors, and assess security issues that may be introduced by vendor customization. Their results show that customization is responsible for a number of security problems ranging from over-privileged to buggy system apps that can be exploited to mount permission redelegation or content leaks attacks. Harehunter [5] reveals a new category of Android vulnerabilities, called Hares, caused by the customization process. Hares occur when an attribute is used on a device but the party defining it has been removed during the customization. A malicious app can "impersonate" the missing attribute to launch privilege escalation, information leakage and phishing attacks. ADDICTED [29] finds that many custom Android devices do not properly protect Linux device drivers, exposing them to illegitimate parties.

All the problems reported so far on Android customization are mainly caused by vendors' altering of critical configurations. They change security configurations of system apps and Linux device drivers; they also remove, add, and alter system apps. Although the existing work has studied several aspects of security problems in the changes of system/app configurations, there is no work that systematically finds all security configuration changes caused by vendor customization, how likely it can lead to security problems, what risky configuration changes are often made by vendors, etc.

In this work, we make the first attempt to systematically detect security configuration changes introduced by parties in the customization chain. Our key intuition is that through comparing a custom device to similar devices from other vendors, carriers, and regions, or through comparing different OS versions, we might be able to find security configuration changes created unintentionally during the customization. More importantly, through a systematic study, we may be able to find valuable insights in vendor customization that can help ven-

dors improve the security of their future customizations. We propose DroidDiff, a tool that detects inconsistent security configurations in a large scale, and that can be employed by vendors to locate risky configurations.

The first challenge that we face in our systematic study is to identify what configurations are security relevant and are likely to be customized. We start from the Android layered architecture and list access control checks employed at each layer. Then, for each check, we rely on Android documentation and our domain knowledge to define corresponding security features. We further analyze how different configurations of these features across custom images can lead to inconsistencies and thus affect the access control check semantics. As a result, we have identified five categories of features. DroidDiff then extracts these features from 591 custom Android ROMs that we collected from multiple sources. This step produces the raw data that will be used for our analysis.

The next challenge is how to compare these images to find out whether they have inconsistent values for the features that we extracted. Given a set of images, conducting the comparison itself is not difficult; the difficulty is to decide the set of images for comparison. If we simply compare all the 591 images, it will not provide much insight, because it will be hard to interpret the implications of detected inconsistencies. To gain useful insights, we need to select a meaningful set of images for each comparison. Based on our hypothesis that inconsistencies can be introduced by vendors, device models, regions, carriers, and OS versions, we developed five differential analysis algorithms: *Cross-Vendor*, *Cross-Model*, *Cross-Region*, *Cross-Carrier*, and *Cross-Version* analyses, each targeting to uncover inconsistencies caused by customization of different purposes. For example, in the Cross-Vendor analysis, we aim to know how many inconsistencies are there among different vendors; in the Cross-Model analysis, we attempt to identify whether vendors may further introduce inconsistencies when they customize Android for different models (e.g. Samsung S4, S5, S6 Edge).

DroidDiff results reveal that indeed the customization process leads to many inconsistencies among security features, ranging from altering the protection levels of permissions, removing protected broadcasts definitions, changing the requirement for obtaining critical GIDs, and altering the protection configuration of app components. We present our discoveries in the paper to show the inconsistency situations among each category of features and how versions, vendors, models, region, and carriers customizations impact the whole situation.

Not all inconsistencies are dangerous, but some changes patterns are definitely risky and warrant further investigation. We have identified such risky patterns, and presented results to show how prevalent they are in the customization process. The inconsistencies expose systems to potential attacks, but if the vendors understand fully the implication of such customization, they will more likely remedy the introduced risks by putting proper protection at some other places. Unfortunately, most of the inconsistencies seem to be introduced by developers who do not fully understand the security implications. Therefore, our DroidDiff can help vendors to identify the inconsistencies introduced during their customization, so they can question themselves whether they have implemented mechanisms to remedy the risks.

To demonstrate that the identified inconsistencies, if introduced by mistakes, can indeed lead to attacks, we picked few cases detected through our differential analysis, and designed proof-of-concept attacks on physical devices[1]. We have identified several real attacks. To illustrate, we found that a detected inconsistency on Nexus 6 can be exploited to trigger emergency broadcasts without the required system permission and another similar one on Samsung S6 Edge allows a non-privileged app to perform a factory reset without a permission or user confirmation. Through exploiting another inconsistency on Samsung Note 2, an attacker can forge SMS messages without the SEND_SMS permission. Moreover, an inconsistency related to permission to Linux GID mapping allows apps to access the camera device driver with a normal protection level permission. We have filed security reports about the confirmed vulnerabilities to the corresponding vendors. We strongly believe that vendors, who have source code and know more about their systems, can find more attacks from our detected risky inconsistencies. We also envision that in the future, vendors can use our proposed tool and database to improve their customization process.

Contributions. The scientific contributions of this paper are summarized as the followings:

- We have systematically identified possible security features that may hold different configurations because of the Android customization process.
- We have developed five differential analysis algorithms and conducted a large-scale analysis on 591 Android OS images. Our results produce significant insights on the dangers of vendor customization.
- We have identified risky configuration inconsistencies that may have been introduced unintentionally during customization. Our results can help vendors' security analysts to conduct further investigation to confirm whether the risks of the inconsistencies are offset in the system or not. We have confirmed via our own attacks that some inconsistencies can indeed lead to actual security breaches.

[1] Due to resource limitation, we could not design the attacks for all the cases identified in our analysis.

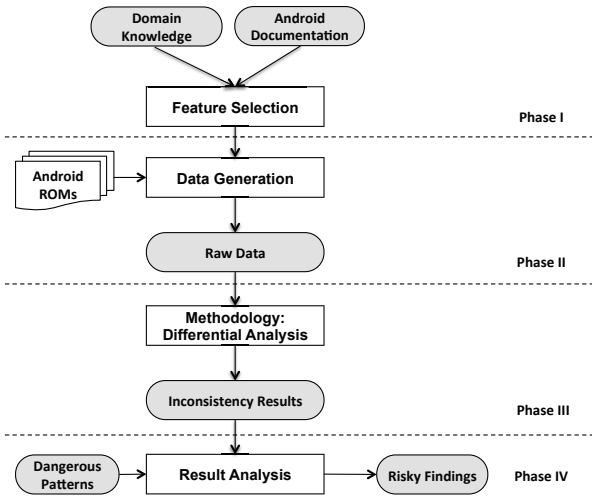


Figure 1: Investigation Flow

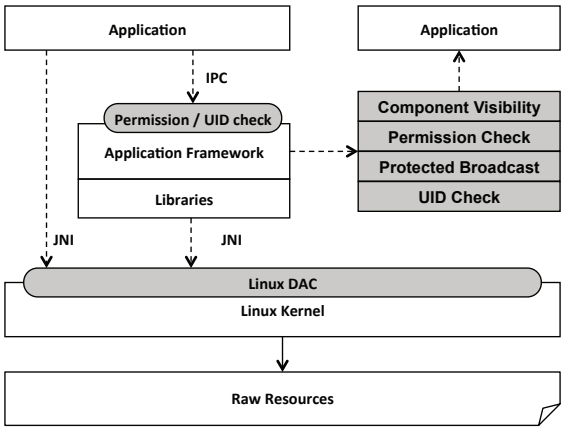


Figure 2: Android Security Model

2 Investigation & Methodology

In this research work, we investigate Android's security features which are configurable during customization at the level of the framework and preloaded apps. Figure 1 depicts our investigation flow. As our work is data driven, the first and second phase are mainly concerned with locating and extracting meaningful security features from our collected Android custom ROMs. The two phases generate a large data set of configurations of the selected security features per image. The third phase performs differential analysis on the generated data according to our proposed algorithms to find any configuration discrepancies. It should be noted that it is out of our scope to find any security feature that is wrongly configured on all images, as obviously, it would not be detected through our differential analysis.

In the last phase, we analyze the detected discrepancies to pinpoint risky patterns. We have confirmed that they are indeed dangerous through high impact attacks. We discuss in the next sections each phase in details.

3 Feature Extraction

In this phase, we aim to extract security features that can cause potential vulnerabilities if altered incautiously during the customization process. To systematically locate these security features, we start from the Android layered architecture (Figure 2) and study the security enforcement employed at each layer.

As Figure 2 illustrates, Android is a layered operating system, where each layer has its own tasks and responsibilities. On the top layer are preloaded apps provided by the device vendors and other third parties such as carriers. To allow app developers to access various resources and functionalities, Android Framework layer provides many high-level services such as Package Manager, Activity Manager, Notification Manager and many others. These services mediate access to system resources and enforce proper access control based on the app's user id and its acquired Android permissions. Additionally, certain services might enforce access control based on the caller's package name or certificate. Right below the framework layer lies the Libraries layer, which is a set of Android specific libraries and other necessary libraries such as libc, SQLite database, media libraries, etc. Just like the framework services, certain Android specific libraries perform various access control checks based on the caller's user id and its permissions as well. At the bottom of the layers is Linux kernel which provides a level of abstraction between the device hardware and contains all essential hardware drivers like display, camera, etc. The Linux kernel layer mediates access to hardware drivers and raw resources based on the standard Discretionary Access Control (DAC).

To encourage collaboration and functionality re-use between apps, Android apps are connected together by Inter-Component Communication (ICC). An app can invoke other apps' components (e.g. activities and services) through the intent mechanism. It can further configure several security parameters to protect its resources and functionalities. As summarized in Figure 2, it can make its components private, require the caller to have certain permissions or to belong to a certain process.

Based on Figure 2, we summarize the Access Control (AC) checks employed by Android in Table 1. We specify the ones whose security features might be altered *statically* during device customization. By *static* modification, we refer to any modification that can be performed through changing framework resources files (including framework-res*.xml which contains most configurations of built-in security features), preloaded apps' manifest files and other system-wide configuration files

(platform.xml and *.xml under /etc/permissions/).

In the following section, we describe in details each configurable AC check and define its security features based on Android documentation and our domain knowledge. We further justify how inconsistent configurations of these features across custom images can bring in potential security risks. Please note that we do not discuss AC checks based on Package Names as previous work [5] has covered the effects of customizing them.

Before we proceed, we present some notations that we will be referring to in our analysis. IMG denotes a set of our collected images. E_P, E_{GID}, E_{PB} and E_C represent a set of all defined permissions, GIDs, protected broadcasts and components on IMG, respectively.

3.1 Permissions

Default and custom Android Permissions are used to protect inner components, data and functionalities. The protection level of a permission can be either `Normal`, `Dangerous`, `Signature`, or `SystemOrSignature`. These protection levels should be picked carefully depending on the resource to be protected. `Signature` and `SystemOrSignature` level permissions are used to protect the most privileged resources and will be granted only to apps signed with the same certificate as the defining app. `Dangerous` permissions protect private data and resources or operations affecting the user's stored data or other apps such as reading contacts or sending SMS messages. Requesting permissions of `Dangerous` levels requires explicit user's confirmation before granting them. `Normal` level on the other hand, is assigned to permissions protecting least privileged resources and do not require user's approval. The following is an example of a permission declaration:

```
<permission android:name="READ_SMS"
     android:protectionLevel="Dangerous">
```

We aim to find if a permission has different protection levels across various images. For example, on vendor A, a permission `READ_A` is declared with `Normal` protection level, while on vendor B, the same permission is declared with a `Signature` one. This would expose the underlying components that are supposed to be protected with more privileged permissions. It would also create a big confusion for developers, as the same permission holds different semantics across images.

Formally, for each defined permission e $\in$ E_P, we define the security feature fn_e as the following:

$$fn_e = ProtectionLevel(e)$$

The potential values of fn_e is in the set {Normal, Dangerous, Signature, Unspecified, 0}. We map `SignatureOrSystem` level to `Signature`, as both of them cannot be acquired by third party apps without a signature check. An `unspecified` value refers to a permission that has been defined without a protection level, while `0` refers to a permission that is not defined on an image.

Table 1: Security Checks

AC Checks	Layer	Configurable
UID	Kernel, Framework Library, App	No
GID	Kernel	Yes
Package Name	Framework, App	Yes
Package Signature	Framework, App	No
Permission	Framework, Library App	Yes
Protected Broadcast	App Layer	Yes
Component Visibility	App Layer	Yes
Component Protection	App Layer	Yes

3.2 GIDs

Certain lower-level Linux group IDs (GIDs) are mapped to Android permissions. Once an app process acquires these permissions, it will be assigned the mapped GID, which will be used for access control at the kernel. Permissions to GID mappings for built-in and custom permissions are defined mostly in platform.xml and other xml files under /etc/permissions/. The following is an example of a permission to GID mapping:

```
<permission android:name =
     "android.permission.NET_TUNNELING">
  <group gid="vpn" />
</permission>
```

In the above example, any process that has been granted `NET_TUNNELING` permission (defined with a `Signature` level) will be assigned the `vpn` GID, and consequently perform any filesystem (read, write, execute) allowed for this GID.

Android states that any change made incautiously to platform.xml would open serious vulnerabilities. In this analysis, we aim to find if the customization parties introduce any modifications to these critical mappings and if so, what damages this might create. More specifically, we want to reveal if vendors map permissions of lower protection levels to existing privileged GIDs, which can result in downgrading their privileges. Following the same example above, assume that on a custom image, the vendor maps a permission `vendor.permission` (defined with `Normal` protection) to the existing `vpn` GID. This new mapping would downgrade the privilege of `vpn` GID on the custom image as it can be acquired with a `Normal` permission instead of a `Signature` one. Thus, any third party app granted `vendor.permission` will run with `vpn` GID

attached to its process, which basically allows it to perform any filesystem permissible for `vpn` GID, usually allowed to only system processes.

To allow discovering vulnerable GID to permission mappings, we extract the minimum permission requirement needed for acquiring a certain GID on a given image; i.e. the minimum protection level for all permissions mapping to it. If the same GID has different minimum requirements on 2 images, then it is potentially vulnerable. For the previous example, we should be able to reveal that `vpn` GID is problematic as it can be acquired with a `Normal` permission level on the custom image and with a `Signature` one on other images.

For each defined GID e $\in E_{GID}$, let P_e denote the permission set mapping to e, we define the feature fn_e:

$$fn_e = GIDProtectionLevel(e),\ where:$$
$$GIDProtectionLevel(e) = \min_{\forall p \in P_e} ProtectionLevel(p)$$

3.3 Protected Broadcasts

Protected broadcasts are broadcasts that can be sent only by system-level processes. Apps use protected broadcasts to make sure that no process, but system-level processes can trigger specific broadcast receivers. System apps can define protected broadcasts as follows:

```
<protected-broadcast android:name="broadcast.name"/>
```

Another app can use the above defined protected-broadcast through the following:

```
<receiver android:name="ReceiverA">
  <intent-filter>
    <action = "broadcast.name"/>
  <intent-filter/>
<receiver/>
```

The above `ReceiverA` can be triggered only by system processes broadcasting `broadcast.name` protected broadcast. The app can alternatively use protected broadcast through dynamically registered broadcast receivers. As it is known, during the customization process, certain packages are removed and altered. We hypothesize that because of this, certain protected broadcasts' definitions will be removed as well. We aim to uncover if these inconsistently non-protected broadcasts are still being used though, as action filters within receivers. This might open serious vulnerabilities, as the receivers that developers assumed to be only invocable by system processes will now be invocable by any third-party app and consequently expose their functionalities.

Formally, for each Protected Broadcast e $\in E_{PB}$, we define the following:

$$fn_e = DefineUse(e),$$

Where DefineUse(e) is defined as the following:

$$DefineUse(e) = \begin{cases} 1 \text{ if } e \text{ is used on an image but not defined} \\ 0 \text{ for other cases} \end{cases}$$

3.4 Component Visibility

Android allows developers to specify whether their declared components (activities, services, receivers and content providers) can be invoked externally from other apps. The visibility can be set through the `exported` flag in the component declaration within the app's manifest file. If this flag is not specified, the visibility will be implicitly set based on whether the component defines intent filters. If existing, the component is exported; otherwise, it is not as illustrated in the following snippet.

```
// Service1 is private to the app
<service android:name="Service1"/>
// Service2 is not private to the app
<service android:name="Service2">
  <intent-filter> ... <intent-filter/>
</service>
```

We would like to uncover any component that has been exposed on one image, but not on another. We assume that if the same component name appears on similar images (e.g. same models, same OS version), then most likely, the component is providing the same functionality or protecting the same data (for content providers). Thus, its visibility should be the same across all images. To account for the cases where a component has been exported but with an added `signature` permission requirement, we consider them as implicitly unexposed.

Formally, for each defined component e $\in E_C$, we extract the following feature:

$$fn_e = Exported(e)$$

The potential values of fn_e is either {true, false, 0}. `0` refers to a non-existing component on a studied image.

3.5 Component Protection

Apps can use permissions to restrict the invocation of their components (services, activities, receivers). In the next code snippet, ServiceA can be invoked if the caller acquires `vendor.permissionA`. Moreover, an app can use permissions to restrict reading and writing to its content provider, as well as to specific paths within it. `android:readPermission` and `android:writePermission` take precedence over `android:permission` if specified, as shown in the code snippet. Components inherit their parents' permission if they do not specify one.

```
<service android:name="ServiceA"
     android:permission="vendor.permissionA"/>
```

```
<provider android:authorities="providerId"
     android:name="providerB"
     android:Permission="vendor.permissionB"
     android:readPermission="vendor.read"
     android:writePermission="vendor.write">
```

We aim to find if the same component has different protection requirements on similar images. Protection mismatch might not necessarily indicate a flaw if the component is not exposed. That's why, we only consider protection mismatches in case of exported components.

We list three cases where a component can be unintentionally exposed on one image, but protected on other images. First is the permission requirement is removed from the component's declaration. Second is the permission protecting it is of lower privilege compared to other images. Third, the permission used is not defined within the image, which makes it possible for any third-party app to define it and consequently invoke the underlying component. To discover components with conflicting protections, we map used permissions to their declarations within the same image. Any mismatch would indicate a possible security flaw for this component.

Formally, let P_e represents the permission protecting a component e $\in E_c$. We define the following feature:

$$fn_e = Protection(e);$$

Where Protection(e) is defined as:

$$Protection(e) = \begin{cases} 0 \text{ if } e \text{ is not defined} \\ 1 \text{ if } P_e \text{ is None; i.e. e is not protected} \\ ProtectionLevel(P_e) \text{ otherwise} \end{cases}$$

In the case where e is a content provider, we define P_{read} and P_{write} representing its read and write permissions and extract fn_e for both cases.

4 Data Generation

To reveal whether customization parties change the configurations of the mentioned security features, we conduct a large scale differential analysis. We collected 591 Android ROMs from Samsung Updates [4], other sources [3, 1, 2], and physical devices. These images are customized by 11 vendors, for around 135 models, 45 regions and 8 carriers. They operate Android versions from 4.1.1 to 5.1.1. Details about the collected images are in Table 2. In total, these images include on average 157 apps per image and 93169 all together apps. To extract the values of the selected security features on each image, we developed a tool called DroidDiff. For each image, DroidDiff first collects its framework resources Apks and preloaded Apks then runs Apktool to extract the corresponding manifest files. Second, it collects configuration files under /etc/permission/. Then,

Table 2: Collected Android Images

Version	# of Distinct Vendors	# of images
Jelly Bean	9	102
KitKat	9	177
Lollipop	8	312
Total	11	591

Table 3: Security Configurations Map

Image	e $\in E_P$ MIPUSH_RECEIVE	e $\in E_{GID}$ camera GID	e $\in E_C$ sms
I1: Xiaomi RedMi 1 Version: 4.4.2	Signature	Normal	True
I2: Xiaomi Mi 2A Version: 4.1.1	Unspecified	Dangerous	False

DroidDiff searches the extracted manifests and configuration files for the definitions of the targeted entities (E_P, E_{PB}, E_{GID} and E_C). Finally, DroidDiff runs the generated values through our differential analysis methodologies, discussed in the next section.

5 Differential Analysis

In our analysis, we aim to detect any feature fn_e having inconsistent values throughout a candidate set of images. Any inconsistency detected indicates a potential unintentional configuration change introduced by a customization party and requires further security analysis to assess possible consequent damages.

Let $fv(fn_e, img)$ represent the value of the feature fn_e on a given image img. To illustrate fn_e to $fv(fn_e, img)$ mappings, consider this real world example depicted in Table 3. As shown, we extract 3 security features and their corresponding values from 2 Xiaomi images. For the custom permission e = `MIPUSH_RECEIVE`, our feature extraction step generates the following values $fv(fn_e, I1)$ = `Signature`, and $fv(fn_e, I2)$ = `Unspecified`.

Let *IMG* denote a set of candidate images to be compared, we define a feature fn_e as inconsistent if:

$$C(fn_e) = \exists x \exists y [x \in IMG \wedge y \in IMG \wedge x \neq y \wedge fv(fn_e, x) \neq fv(fn_e, y)]$$

The above statement means that we consider the feature fn_e inconsistent across the set IMG if there exists at least two different images where the value of fn_e is not equal. It should be noted that we do not consider any cases where $fv(fn_e, img)$ = `0` for e $\in \{E_P, E_{GID}$ and $E_C\}$.

Sample Selection. To discover meaningful inconsistencies through differential analysis, our collected images should be clustered based on common criteria. A meaningful inconsistency would give us insights about the responsible party that introduced it. For example,

to reveal if inconsistencies are introduced by an OS upgrade, it would not make sense to select images from all vendors, as the inconsistency could be due to customizing the device for a specific vendor, rather than because of the OS upgrade. Similarly, to uncover if a specific vendor causes inconsistencies in a new model, it is not logical to compare it with models from other vendors. Rather, we should compare it with devices from the same vendor. Besides, to avoid detecting a change caused by OS version mismatches, the new model should be compared to a model running the same OS version.

We designed five different algorithms that target to uncover meaningful inconsistencies. Specifically, by carefully going through each party within the customization chain, we designed algorithms that would reveal inconsistencies (if any) caused by each party. Further, for each algorithm, we select our candidate images based on specific criteria that serve the purpose of the algorithm,

We describe each algorithm as well as the sample selection criteria in the next sections.

A1: Cross-Version Analysis. This analysis aims to uncover any inconsistent security features caused by OS version upgrades. We select candidate image sets running similar device models to make sure that the inconsistency is purely due to OS upgrade. For instance,we would pick 2 Samsung S4 devices running 4.4.4 and 5.0.1 as a candidate image set, and would reveal if upgrading this model from 4.4.4 to 5.0.1 causes any security configuration changes. Formally, let $\text{IMG}_{\text{MODEL}}$ denote the candidate image set as the following:

$$IMG_{\text{MODEL}} = \{img_1, img_2, ..., img_n\}$$
$$\text{such that } img_i \in IMG_{MODEL} \text{ if } model(img_i) = MODEL$$

Based on our collected images, this algorithm generated 135 candidate image sets (count of distinct model).

Let $fv(fn_e, img)$ denote a value for a feature fn_e in img $\in$ IMG_{MODEL}. We define the inconsistency condition under Cross-Version analysis algorithm as follows,

$$\begin{aligned} C_{Version}(fn_e) = \exists x \exists y [\, & x \in IMG_{MODEL} \wedge y \in IMG_{MODEL} \\ & \wedge x \neq y \wedge fv(fn_e, x) \neq fv(fn_e, y) \\ & \wedge version(x) \neq version(y)] \end{aligned}$$

The above condition implies that fn_e is inconsistent if there exist two same model images running different versions, and where the values of fn_e is not the same. DroidDiff runs the analysis for each of the 135 candidate sets and generate the number of inconsistencies detected.

A2: Cross-Vendor Analysis. This analysis aims to reveal any feature fn_e that is inconsistent across vendors. To make sure that we are comparing images of similar criteria across different vendors, we pick candidate image sets running the same OS version (e.g. HTC M8 and Nexus 6 both running 5.0.1). Our intuition here is that if an inconsistency is detected, then the vendor is the responsible party. We formally define the candidate image set as the following:

$$IMG_{\text{VERSION}} = \{img_1, img_2, ..., img_n\}$$
$$\text{such that } img_i \in IMG_{VERSION} \text{ if } version(img_i) = VERSION$$

This algorithm generated 12 candidate image sets (count of distinct OS versions that we collected).

Let $fv(fn_e, img)$ denote a value for a feature fn_e in img $\in$ $IMG_{VERSION}$. We redefine the inconsistency condition under Cross-Vendor analysis as follows:

$$\begin{aligned} C_{Vendor}(fn_e) = \exists x \exists y [\, & x \in IMG_{VERSION} \wedge y \in IMG_{VERSION} \\ & \wedge x \neq y \wedge fv(fn_e, x) \neq fv(fn_e, y) \\ & \wedge vendor(x) \neq vendor(y)] \end{aligned}$$

The last condition implies that fn_e is inconsistent if there exists two images from different vendors, but running the same OS version, where its value is not equal.

A3: Cross-Model Analysis. In this analysis, we want to uncover any feature fn_e that is inconsistent through different models. For example, we want to compare the configurations on Samsung S5 and Samsung S4 models, running the same OS versions. To ascertain that any inconsistency is purely due to model change within the same vendor, we pick our candidate image sets running the same OS version, defined as $\text{IMG}_{\text{VERSION}}$ in the previous example. We further make sure that we are comparing models from the same vendor by adding a new check in the next condition.

Let $fv(fn_e, img)$ denote a value for fn_e in img $\in$ $IMG_{VERSION}$. We redefine the inconsistency condition under Cross-Model analysis as follows:

$$\begin{aligned} C_{Model}(fn_e) = \exists x \exists y [\, & x \in IMG_{VERSION} \wedge y \in IMG_{VERSION} \\ & \wedge x \neq y \wedge fv(fn_e, x) \neq fv(fn_e, y) \\ & \wedge vendor(x) = vendor(y) \wedge model(y) \neq model(x)] \end{aligned}$$

The last condition implies that fn_e is inconsistent if there exists two images from the same vendor, running the same OS version, but customized for different models, where its value is not equal.

A4: Cross-Carrier Analysis. We aim to uncover any inconsistent security features fn_e through different carriers (e.g., a MotoX from T-Mobile, versus another one from Sprint). To make sure that we are comparing images running the same OS version, we pick our candidate image sets from $\text{IMG}_{\text{VERSION}}$. We further make sure that we are comparing images running the same model

as shown in the following inconsistency condition:

$$C_{Carrier}(fn_e) = \exists x \exists y [x \in IMG_{VERSION} \wedge y \in IMG_{VERSION} \wedge x \neq y \wedge fv(fn_e, x) \neq fv(fn_e, y) \wedge carrier(x) \neq carrier(y) \wedge model(y) = model(x)]$$

The last conditions in the above definition of $C_{Carrier}$ implies that fn_e is inconsistent if there exists two images running the same model and OS versions, but from different carriers where its value is not the same.

A5: Cross-Region Analysis. This analysis intends to find any inconsistencies in the configuration of security features fn_e through different regions (e.g. LG G4, Korean edition versus US edition). Any inconsistencies detected will be attributed to customizing a device for a specific region. We pick our candidate image sets from $IMG_{VERSION}$ to make sure that we are comparing images running the same OS version. We define the inconsistency count under Cross-Carrier analysis as follows:

$$C_{Region}(fn_e) = \exists x \exists y [x \in IMG_{VERSION} \wedge y \in IMG_{VERSION} \wedge x \neq y \wedge fv(fn_e, x) \neq fv(fn_e, y) \wedge region(x) \neq region(y) \wedge model(y) = model(x)]$$

The last conditions in the above definition of C_{Region} implies that fn_e is inconsistent if there exists two images running the same model and OS versions, but from different regions where its value is not the same.

6 Results and Findings

We conduct a large-scale differential analysis on our collected images using the aforementioned methodologies with the help of DroidDiff. The analysis discovered a large number of discrepancies with regards to our selected features. In this section, we present the results and findings.

6.1 Overall Results

Figure 3 shows the overall changes detected from our analysis. We plot the average percentage of inconsistencies detected for each feature category using the five differential analysis algorithms. To provide an estimate of the inconsistencies count, each box plot shows an average number of total common entities (appearing on at least 2 images) in the image sets studied; we depict this number as # total in the graph. Let us use the first box plot as an example to illustrate what the data means: under the Cross-Version analysis (A1), DroidDiff generated on average 673 common permissions per each studied candidate sets. 50% of the candidate image sets contain at least 4.8% of total permissions (around 32 out of 673) having inconsistent protection levels; those in the top 25 percentile (shown in the top whisker) have at least 6% (40) inconsistent permissions. Figure 3 also depicts the image sets that are outliers, i.e., they have particularly higher number of inconsistencies compared to the other image sets in the same group. For instance, the candidate image set $IMG_{Version=4.4.2}$ in the Cross-Vendor analysis (A2) contains around 10% of GIDs whose protections are inconsistent.

As depicted in Figure 3, the Cross-Version analysis (A1) detects the highest percentage of inconsistencies in all 5 categories, which means that upgrading the same device model to a different OS version introduces the highest security configuration changes. An intuitive reason behind this is that through a new OS release, Android might enforce higher protections on the corresponding entities to fix some discovered bugs (e.g. adding a permission requirement to a privileged service). However, we found out that through newer OS releases, certain security features are actually downgraded, leading to potential risks if done unintentionally. We discuss this finding in more details in Section 6.6.

Through the Cross-Vendor analysis (A2), DroidDiff detects that several security features are inconsistent among vendors, even though they are of the same OS version. We have further analyzed the vendors that cause the highest number of inconsistencies. An interesting observation is that smaller vendors, such as BLU, Xiaomi and Digiland caused several risky inconsistencies. In fact, all inconsistent GIDs are caused by these 3 companies. Probably, small vendors may not have enough expertises to fully evaluate the security implications of their actions.

The Cross-Model analysis (A3) also detects a number of inconsistencies, which means that different device models from the same vendor and OS version, might have different security configurations.

Although the Cross-Carrier (A4) and Cross-Region (A5) analyses detect a smaller percentage of inconsistencies, it is still significant to know that the same device model running the same OS version might have some different configurations if it is customized for different carriers or regions. Our results shows that the inconsistencies are less common in North America region, and more prevalent in Chinese editions.

6.2 Permissions Changes Pattern

Protection level mismatch. DroidDiff results confirm that Android permissions may hold different protection levels across similar images. As Figure 3 illustrates, more than 50% of the candidate image sets contain at least 32 (out of 673), 9 (out of 817) permissions having inconsistent protection levels in the Cross-Version (A1) and Cross-Model (A3) analyses, respectively. To reveal

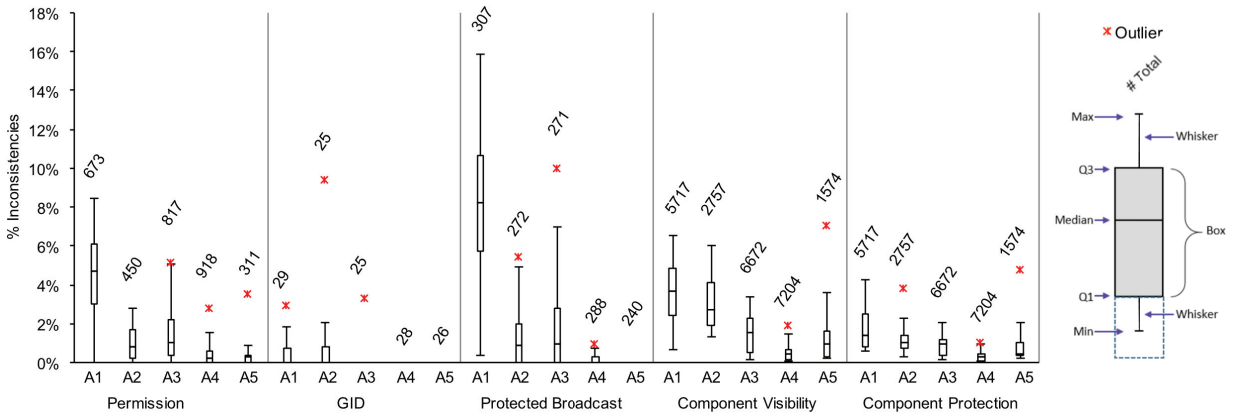


Figure 3: Overall Inconsistencies Detected
A1: Cross-Version, A2: Cross-Vendor, A3: Cross-Model, A4: Cross-Carrier, A5: Cross-Region

more insights, we checked which combination of protection level changes are the most common. That is, which combination out of the following 3 possible combinations is the most common (`Normal`, `Dangerous`), (`Normal`, `Signature`) or (`Dangerous`, `Signature`). We have calculated the occurrence of each pattern, and present the results in Figure 4. As shown, (`Normal`, `Signature`) combination is the most common pattern. This is quite serious as several permissions that hold a `Signature` protection level on some images are defined with a `Normal` protection level on others. We present here two permissions holding inconsistent protection levels:

- `com.orange.permission.SIMCARD_AUTHENTICATION` holds `Signature` and `Normal` protection on Samsung S4(4.2.2) and Sony Experia C2105 (4.2.2), respectively.
- `com.sec.android.app.sysscope.permission.RUN_SYSSCOPE` holds `Dangerous` and `Signature` protection on Samsung Note4 (5.0.1) and S4(5.0.1).

Usage of unspecified protection level. Android allows developers to define a permission without specifying a protection level, in which case, the default protection level is `Normal`. In our investigation, we found that it is not clear whether developers really intended to use `Normal` as the protection level. We found that a large percentage of these permissions (with unspecified protection level) hold conflicting protections on other images. Overall, 2% of the permissions studied were defined without a specified protection level in at least one image. To check if developers intended to use `Normal` as the protection level, for each permission that has been defined without a protection level, we check its corresponding definitions on other images to see if it has a protection level specified. We then compare the other specification to see it it is `Normal` or not. As Figure 5(a) illustrates, on average, 91% of these permissions holding

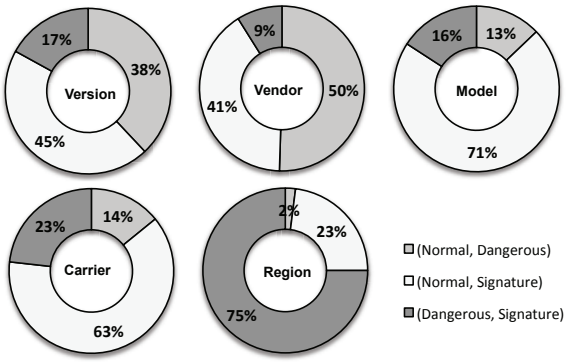


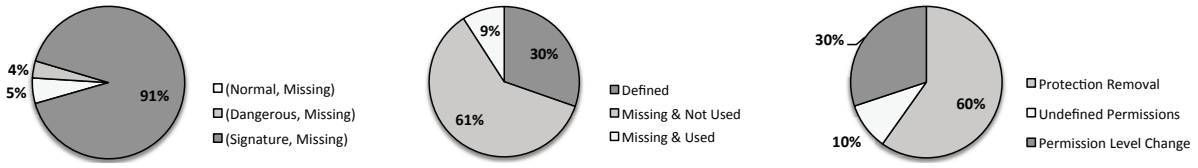
Figure 4: Protection Level Changes Patterns

unspecified protection level hold a `Signature` protection on at least 1 other image, which indicates that developers probably intended to use the `Signature` protection level. We illustrate this finding with 2 permissions:

- `com.sec.android.phone.permission.UPDATE_MUTE_STATUS` holds `Unspecified` and `Signature` protections on Samsung E7 (5.1.1) and S6 Edge(5.1.1), respectively.
- `com.android.chrome.PRERENDER_URL` holds `Unspecified` and `Signature` protections on LG Vista (4.4.2) and Nexus7 (4.4.2), respectively.

6.3 Permission-GID Mapping

By analyzing the differential analysis results of the mappings between GIDs and permissions, we have confirmed that customization introduces problematic GID-to-permission mappings that can lead to serious vulnerabilities in the victim images. Through the Cross-Vendor analysis (A2), DroidDiff detects 3 inconsistent cases (out of 25 common GIDs), in which vendors mapped less privileged permissions to privileged GIDs. This dangerous pattern leads to downgrad-



(a) Unspecified Protection Level Change Patterns

(b) Protected Broadcast Inconsistencies

(c) Causes of Components Protection Mismatch

Figure 5: Inconsistency Breakdown

ing the protection level of these GIDs. We illustrate this finding with one detected example. On AOSP images and several customized images (running 4.4.4 and below), `camera` GID is mapped to a `Dangerous` level permission (`android.permission.CAMERA`). However, on Neo 4.5 (BLU), we found out that the same GID is mapped to a `Normal` level permission: `android.permission.ACCESS_MTK_MMHW`. This case indicates that BLU has downgraded the requirement for apps to obtain the `camera` GID. Our analysis reveals that the requirements for two more GIDs, `system` GID and `media` GID, have been downgraded. These two GIDs, protected by a `Signature` permission on most devices, can be acquired with a `Normal` permission on the victim devices.

6.4 Protected Broadcasts Changes Pattern

DroidDiff further reveals that protected broadcasts' definitions might be removed from some images during the customization process. As illustrated in Figure 5(b), through the Cross-Version analysis (A1), we detected that 70% of protected broadcast are not defined on at least one vendor. This might not necessarily be problematic if the broadcast is not used. However, our investigation shows that around 9% of these inconsistently unprotected broadcasts (28 on average per image set) are used as intent-filters actions for broadcast receivers. This inconsistency across versions is quite alarming as a privileged receiver that was supposed to be invoked by system processes can be invoked by any unprivileged app on certain versions. As Figure 3 further illustrates, Cross-Vendor (A2) and Cross-Model (A3) analyses reveal that more than 25% of candidate image sets contain at least 2% broadcasts which are inconsistently protected, but still being used as intent-filter actions.

6.5 Component Security Changes Pattern

Visibility mismatch. DroidDiff results confirm that app components may have a conflicting visibility. That is, the component is exposed on one image but not on another. As Figure 3 illustrates, 50% of the candidate image sets contain at least 3.9% components (around 222) and 2% (133) holding inconsistent visibility through various versions (A1) and models (A3), respectively. To provide insights about which components hold more visibility inconsistencies, we break down our findings to activities, services, receivers, and content providers. We plot the results in Figure 6. As depicted, content providers and activities have the highest visibility mismatch. In fact, 25% of the candidate image sets contain at least 20% (53) and 14% (21) content providers holding a different visibility in different versions (A1) and vendors (A2), respectively. Similarly, 4% (139) and 3% (45) of activities hold a conflicting visibility in 50% of the studied sets based on A1 and A2, respectively.

Permission mismatch. DroidDiff further reveals that components may hold inconsistent protections across images. We break down our findings in Figure 8 (see appendix). Our results show that content providers exhibit the highest number of protection inconsistencies. In fact, more than 25% of the candidate images sets include at least 19% (51) and 10% (33) content providers having different protections in the Cross-Version (A1) and Cross-Model (A3) analyses, respectively. We have further analyzed these inconsistent components and categorized the reason behind the discrepancies. As Figure 5(c) illustrates, in the majority of the cases (60%), the discrepancy is caused by the same component being protected with a permission on one image, but not protected at all on others. The second common reason (30%) is that the same component is protected with permissions holding different protection levels across the studied images. Using non-defined permissions to protect a component is third common reason (10%).

Duplicate components declaration. Based on our analysis of the inconsistent broadcast receivers (particularly high on Lollipop images), we found out that most of them are caused by a non-safe practice that developers follow. Developers declare duplicate broadcast receivers names in the same app, but assign them different protections. After further investigation, we found out that it is not a safe practice to do as it will be possible to bypass any restrictions put on the first defined receiver. To illustrate, consider the following receivers, defined in Samsung's preloaded PhoneErrorService app:

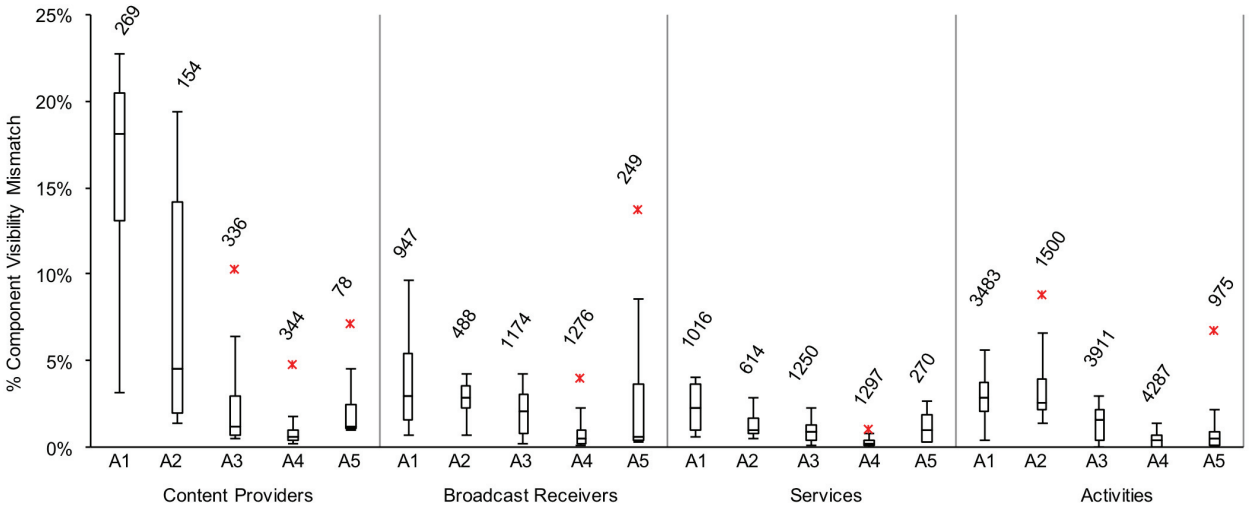


Figure 6: Breaking Down Components: Visibility Mismatch

```
<receiver android:name="PhoneErrorReceiver"
  android:permission="android.permission.REBOOT">
    <intent-filter>
        <action android:name="REFRESH_RESET_FAIL"/>
        ...
    </intent-filter>
</receiver>
<receiver android:name="PhoneErrorReceiver">
     <intent-filter>
        <action
            android:name="DATA_ROUTER_DISPLAY"/>
     </intent-filter>
</receiver>
```

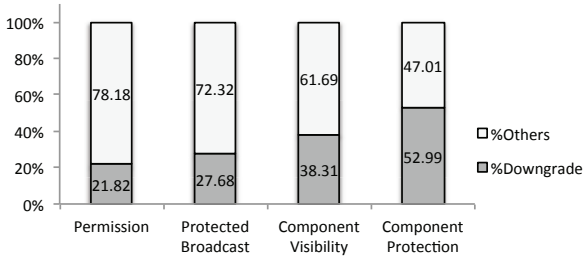


Figure 7: Percentage of Security Features Downgrades

In the above code, the developer decided to protect the functionality triggered when receiving the action `REFRESH_RESET_FAIL` with the permission `REBOOT` (`Signature` level). In the other case, she decided not to require any permissions when invoking the functionality triggered by the action `DATA_ROUTER_DISPLAY`. At first glance, the above duplicate components declaration might look fine. However, we found out that the PackageManagerService does not carefully handle the registration of duplicate receivers. On one hand, it correctly handles mapping each filter to the required permission, used for **implicit** intents routing (e.i., sending the action `REFRESH_RESET_FAIL` requires `REBOOT` permission, while sending `DATA_ROUTER_DISPLAY` does not require any permission). On the other hand, however, it does not correctly map each component name to the required permission, used for **explicit** intents routing (e.i., the first `PhoneErrorReceiver` should require `REBOOT` while the second one should not). In fact, it turns out that the second declaration of the component name replaces the first one. Thus, any protection requirement on the second receiver would replace the first receiver's permission requirement in case of **explicit** invocation. Consequently, in the above example, invoking `PhoneErrorReceiver` explicitly does not require any permission. The explicit intent can further set the action `REFRESH_RESET_FAIL` and thus trigger the privileged functionality (rebooting the phone) without the required `REBOOT` permission. We have confirmed this dangerous pattern in several preloaded apps and were able to achieve various damages. We filed a bug report about this discovered vulnerability to Android Security team and informed other vendors about it.

6.6 Downgrades Through Version Analysis

A dangerous pattern that we are interested in is whether there are any security downgrades through versions. For example, unlike a security configuration upgrade, possibly attributed to fixing discovered bugs in earlier images, downgrading a security configuration is quite dangerous as it will lead to a potential exposure of privileged resources that were already secured on previous versions. For each security configuration, we report in Figure 7, the percentage of security configuration downgrades out of all detected cases. As Figure 7 illustrates, a large number of configurations are indeed downgraded. For example, 52% of inconsistent component protection mismatch are actually caused by downgrading the protection.

7 Attacks

We would like to find out whether the risky patterns discovered can actually lead to actual vulnerabilities. To do that, we have selected some high impact cases, and tried to design attacks to verify whether these cases can become vulnerabilities. Due to resources limitations, our verification is driven by the test devices that we have, including Samsung Edge 6 Plus (5.1.1), Edge 6 (5.0.1), Nexus 6 (5.1.1), Note2 (4.4.2), Samsung S4 (5.0.1), MotoX (5.0.1), BLU Neo4 (4.2.2), and Digiland DL700D (4.4.0). We have found 10 actual attacks, some of which were confirmed on several devices. We have filed security reports for the confirmed vulnerabilities to the corresponding vendors. We discuss here 6 attacks. At the end of this section, we discuss possible impacts of 40 randomly selected cases in other devices to demonstrate the significance of inconsistent security configurations.

Stealing emails. SecEmailSync.apk is a preloaded app on most Samsung devices. It includes a content provider, called `"com.samsung.android.email.otherprovider"`, which maintains a copy of user's emails received through the default Samsung email app. Our Cross-Model and Cross-Region analyses reveal inconsistent permission protections on this provider among several Samsung images. The `Read` and `Write` accesses to this provider are protected with a `Signature` permission `"com.samsung.android.email.permission.ACCESS_PROVIDER"` on Samsung Grand On(5.1.1, India), S6 Edge (5.1.1, UAE), and other devices. However, this provider is not protected with any permission on several other devices such as our test device S6 Edge (5.1.1, Global edition). We wrote an attack app that queries this content provider. It was able to access user's private emails on the victim device without any permission.

Forging premium SMS messages. The TeleService package (`com.android.phone`) is preloaded on many Samsung devices, and provides several services for phone and calls management. A notable service is `.TPhoneService`, which performs some major phone functionalities such as accepting voice and video calls, dialing new phone numbers, sending messages (e.g. to inform why a call cannot be received), as well as recording voice and video calls. Our Cross-Model and Cross-Version analyses reveal a permission mismatch on this critical service. On several devices, such as Samsung S5 LTE-A (4.4.2, Korea), the access to this service is protected with the `Signature` permission `com.skt.prod.permission.OEM_PHONE_SERVICE`, which makes the service unaccessible to third-party apps. However, on several other devices such Samsung Note 2 (4.4.2, Global edition), this service is protected with another permission `com.skt.prod.permission.PHONE_SERVICE` for which our analysis reveals a missing definition. We built an attack app that defines the missing permission with a `Normal` protection level. Our app was able to successfully bind to `com.android.phone.TPhoneService` and invoke the send-message API on Samsung Note 2, allowing to forge SMS messages without the usually required `SEND_SMS`.

Unauthorized factory reset. The preloaded Samsung app ServiceModeApp_FB.apk performs various functionalities related to sensitive phone settings. It includes a broadcast receiver `ServiceModeAppBroadcastReceiver` that listens to several intent filters including the action filter `com.samsung.intent.action.SEC_FACTORY_RESET_WITHOUT_FACTORY_UI` that allows to factory reset the phone and delete all data without user confirmation. Our Cross-Version analysis reveals a protection mismatch for this critical broadcast receiver. In most devices running Kitkat and below, this receiver is protected with the `Signature` permission `com.sec.android.app.servicemodeapp.permission.KEYSTRING`. However, on several Lollipop images, it is not correctly protected. Further investigation reveals that this is caused by the duplicate receiver pattern discussed in Section 6.5. The declaration of the receiver has been duplicated on the victim images such that the first one requires a `Signature` permission while the second one does not. As discussed in Section 6.5, using this risky pattern allows a caller app to bypass any restrictions on the first declared broadcast receivers through explicit invocation. We wrote an attacking app that invokes the broadcast receiver explicitly with the action `com.samsung.intent.action.SEC_FACTORY_RESET_WITHOUT_FACTORY_UI` and were able to factory reset several victim devices including the latest S6 Edge Plus 5.1.1, S6 Edge 5.0.1, and S4 5.0.1.

Accessing critical drivers with a normal permission. Our Cross-Vendor analysis reveals a critical protection downgrade of the `system` GID. On some images, such as Samsung S5 (4.4.2), this GID is mapped to the `Signature` permission `com.qualcomm.permission.IZAT`. Nevertheless, on other images (e.g., Redmi Note 4.4.2 and Digiland DL700D 4.4.0), this GID is mapped to a `Normal` level permission `android.permission.ACCESS_MTK_MMHW`, indicating that any third-party app can easily get the `system` GID. Table 4 lists the device drivers that are accessible via the `system` GID on the Digiland DL700D Tablet. These are privileged drivers, but they can now be accessible to normal apps.

Triggering emergency broadcasts without permission. CellBroadcastReceiver is a preloaded Google

Table 4: Drivers accessible to System GID

Driver	ACL
bootimg; devmap; mtk_disp; pro_info; preloader; recovery	r –
pro_info; devmap; dkb; gps; gsensor; hdmitx; hwmsensor; kb; logo; misc; misc-sd; nvram; rtc0; sec; seccfg ; stpwmt touch; ttyMT2 ; wmtWifi; wmtdetect	rw-
cpuctl	r-x

app that performs critical functionalities based on received cell broadcasts. It registers the broadcast receiver `PrivilegedCellBroadcastReceiver` that allows receiving emergency broadcasts from the cell providers (e.g., evacuation alerts, presidential alerts, amber alerts, etc.) and displaying corresponding alerts. This critical functionality can be triggered if the action `android.provider.Telephony.SMS_EMERGENCY_CB_RECEIVED` is received. Our Cross-Vendor and Cross-Version analyses discovered a protection mismatch on this receiver among several devices. For instance, on Nexus S 4G 4.1.1, this receiver is protected with the `Signature` permission `android.permission.BROADCAST_SMS`. However, on other devices (e.g., Nexus6 5.1.1 and MotoX XT1095 5.0.1), it is protected with the `Dangerous` permission `android.permission.READ_PHONE_STATE`. Our investigation reveals that this is also due to the duplicate receivers risky pattern (Section 6.5). On the victim devices, `PrivilegedCellBroadcastReceiver` has been declared twice such that its first declaration requires a `Signature` permission and handles the action `android.provider.Telephony.SMS_EMERGENCY_CB_RECEIVED`, while the second declaration handles less privileged actions and requires a `Dangerous` permission. As discussed, any third-party app can bypass the permission requirement on the first receiver through explicit invocation. We wrote an attack app that was able to trigger this receiver and show various emergency alerts.

Tampering with system wide settings. SystemUI is a preloaded app that controls system windows. It handles and draws a lot of system UIs such as top status bar, system notification and dialogs. To manage the top status bar, the custom Samsung SystemUI includes a service `com.android.systemui.PhoneSettingService`, which handles incoming requests to turn on/off a variety of system wide settings appearing on the top status bar. These settings include turning on/off wifi, bluetooth, location, mobile data, nfc, driving mode, etc; that are usually done with user consent. Our analysis shows a protection mismatch for this service. On S5(4.4.2) and Note8(4.4.2), this service is protected with a signature permission `com.sec.phonesettingservice.permission.PHONE_SETTING`, while on Note 2, 4.4.2, the service is not protected with any permission. We wrote an attack app that successfully asks the privileged service to turn on all the settings mentioned above without any permission.

Other Randomly Selected Cases. The impact of inconsistent security configurations are significant. In addition to end-to-end attacks we built, we also randomly sampled 40 inconsistencies and manually analyzed what could happen once they were exploited. Note that due to the lack of physical devices, all we could do is just static analysis to infer possible consequences once an exploit succeeds. Such an analysis may not be accurate, but it is still important for understanding the impacts of inconsistent security configurations. The outcomes of our analysis are shown in Table 5. Please note that we could not assess the impact in 5 cases (heavily obfuscated code), while we confirmed that 2 cases have been hardened via runtime checks.

8 Limitations

In this section, we discuss some limitations of our proposed approach.

Components implementation changes. A static change of a component's security configurations (visibility or permission protection) might not necessarily indicate a security risk all the time. In fact, a developer might *intentionally* decide to export a component or downgrade its permission protection in the following cases: the component's operations or supplied data are not privileged anymore or the component's implementation is hardened via runtime checks of the caller's identity (e.g., binder.getCallingUid() or Context.checkPermission() APIs). Our solution pinpoints these possibly *unintentional* risky configurations changes and demands further investigation to confirm whether the change was indeed intentional or not.

Components renaming. Our approach would miss detecting inconsistent configurations of components which have been renamed during the customization. In fact, as Android relies heavily on implicit intents for inter-app communication, vendors might rename their components to reflect their organization identity.

9 Related Work

Security risks in Android customization. The extensive Android vendor customization have been proven to be problematic in prior studies. At the Kernel level, ADDICTED [29] finds under-protected Linux device drivers on customized ROMs by comparing them with their counterparts on AOSP images. Our finding on inconsistent GID to permission mappings demonstrates another

Table 5: Impact of Inconsistent Security Configurations

Inconsistent Configuration Category	Impact	Specific Examples
Permission Protection Change	Change System / App Wide Settings	Xiaomi Cloud Settings, Activate SIM
Removed Protected Broadcasts	Trigger Dangerous Operations and events	Trigger data sync, SMS received Airplane mode active, SIM is full
Non-Protected Content Providers	Data Pollution	Write to system logs, Add contacts Change instant messaging configurations
Non-Protected Content Providers	Data Leaks	Read emails, Read contacts Read blocked contact lists
Non-Protected Services	Trigger Dangerous Operations	Access Location, Bind to printing services Kill specific apps, Trigger backup
Non-Protected Activities	Change System wide Settings	Change Telephony settings, Access hidden activities
Non-Protected Receivers	Trigger Dangerous Operations	Send SMS messages, Trigger fake alerts Alter telephony settings , Issue SIM commands

way that can expose critical device drivers. At the framework/ app level, Harehunter [5] reveals the Hanging Attributes References (Hares) vulnerability caused by the under-regulated Android customization. The Hare vulnerability happens when an attribute is used on a device but the party defining it has been removed. A malicious app can then fill the gap to acquire critical capabilities, by simply disguising as the owner of the attribute. Previous works [13, 14, 25] have also highlighted security issues in the permission and components AC in preloaded apps. Gallo et al [13] analyzed five different devices and concluded that serious security issues such as poorer permission control grow sharply with the level of customization. Other prominent work [25] analyzes the pre-installed apps on 10 factory images and reports the presence of known problems such as over-privilege [11], permission re-delegation [12], etc. Our study is fundamentally different from the above work [25] which finds specific known vulnerabilities on a customized image through conducting a reachability analysis from an open entry point to privileged sinks. Instead, we leverage a differential analysis to point out inconsistencies in components' protection, and consequently detect unintentionally exposed ones. Our analysis further gives insights about possible reasons behind the exposure.

Demystification of Android security configurations. The high flexibility of Android's security architecture demands a complete understanding of configurable security parameters. Stowaway [11] and PScout [7] lead the way by mapping individual APIs to the required permission. Understanding these parameters provides the necessary domain knowledge in our feature selection. This understanding has inspired other researchers to detect vulnerabilities in apps. The prevalence of misconfigured content providers, activities and services is studied in [30, 8], respectively. These vulnerabilities are due to developers' exposing critical components or misinterpreting Android's security protection. Instead of focusing on analyzing an individual app to find if it is vulnerable, our approach learns from the configurations of the same app on other ROMs to deduct if it should be protected or not.

Android vulnerability analysis. Prior research has also uncovered security issues rooted in non-customized AOSP images. PileUp [26] brings to attention the problematic Android upgrading process. Two recent studies examine the crypto misuse in Android apps [9, 16]. Other works evaluate the security risks resulting from design flaws in the push-cloud messaging [18], in the multi-user architecture [24], in Android app uninstallation process [28] and in Android's Clipboard and sharing mechanism [10]. Other researchers [20, 15] focused on uncovering vulnerabilities within specific Android apps in the web landscape. These vulnerabilities are complementary to the security issues detected in vendor customization, and jointly present a more complete picture of Android ecosystem's security landscape. To analyze Android vulnerabilities, static and dynamic analysis techniques have been proposed to address the special characteristics of Android platform. CHEX [19], Epicc [21], and FlowDroid [6] apply static analysis to perform vulnerability analysis. Other works [23, 22, 17, 27] employ dynamic analysis to accurately understand app's behaviors. Both techniques are beneficial to our research. Dynamic analysis can help us exploit the likely risky inconsistencies, while static analysis can bring the control/data flow of framework/ app code as another security feature into our differential analysis. We will explore these ideas in future work.

10 Conclusion

In this paper, we make the first attempt to systematically detect security configuration changes introduced by Android customization. We list the security features applied at various Android layers and leverage differential analysis among a large set of custom ROMs to find out if they are consistent across all of them. By comparing security configurations of similar images (from the same vendor, running the same OS version, etc.), we can find critical security changes that might have been unintentionally introduced during the customization. Our analysis shows that indeed, customization parties change several config-

urations that can lead to severe vulnerabilities such as private data exposure and privilege escalation.

11 Acknowledgement

We would like to thank our anonymous reviewers for their insightful comments. This project was supported in part by the NSF grant 1318814.

References

[1] Android Revolution. http://goo.gl/MVigfq.

[2] Factory Images for Nexus Devices. https://goo.gl/i0RJnN.

[3] Huawei ROMs. http://goo.gl/dYPTE5.

[4] Samsung Updates. http://goo.gl/RVU84V.

[5] AAFER, Y., ZHANG, N., ZHANG, Z., ZHANG, X., CHEN, K., WANG, X., ZHOU, X., DU, W., AND GRACE, M. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS '15.

[6] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. PLDI '14.

[7] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM.

[8] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM.

[9] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM.

[10] FAHL, S., HARBACH, M., OLTROGGE, M., MUDERS, T., AND SMITH, M. Hey, you, get off of my clipboard. In *In proceeding of 17th International Conference on Financial Cryptography and Data Security* (2013).

[11] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM.

[12] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[13] GALLO, R., HONGO, P., DAHAB, R., NAVARRO, L. C., KAWAKAMI, H., GALVÃO, K., JUNQUEIRA, G., AND RIBEIRO, L. Security and system architecture: Comparison of android customizations. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2015).

[14] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)* (Feb. 2012).

[15] JIN, X., HU, X., YING, K., DU, W., YIN, H., AND PERI, G. N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA), CCS '14, ACM.

[16] KIM, S. H., HAN, D., AND LEE, D. H. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM.

[17] KLIEBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (2014), SOAP '14.

[18] LI, T., ZHOU, X., XING, L., LEE, Y., NAVEED, M., WANG, X., AND HAN, X. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS '14, ACM.

[19] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), CCS '12.

[20] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on webview in the android system. ACSAC '11.

[21] OCTEAU, D., MCDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security* (2013), SEC'13.

[22] POEPLAU, S., FRATANTONIO, Y., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. NDSS 14'.

[23] RASTOGI, V., CHEN, Y., AND ENCK, W. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2013), CODASPY '13.

[24] RATAZZI, P., AAFER, Y., AHLAWAT, A., HAO, H., WANG, Y., AND DU, W. A systematic security evaluation of Android's multi-user framework. In *Mobile Security Technologies (MoST) 2014* (San Jose, CA, USA, 2014), MoST'14.

[25] WU, L., GRACE, M., ZHOU, Y., WU, C., AND JIANG, X. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security* (New York, NY, USA, 2013), CCS '13, ACM.

[26] XING, L., PAN, X., WANG, R., YUAN, K., AND WANG, X. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014), SP '14.

[27] YAN, L. K., AND YIN, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), Security'12.

[28] ZHANG, X., YING, K., AAFER, Y., QIU, Z., AND DU, W. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS* (2016).

[29] ZHOU, X., LEE, Y., ZHANG, N., NAVEED, M., AND WANG, X. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA*.

[30] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in android applications. In *NDSS* (2013).

12 Appendix

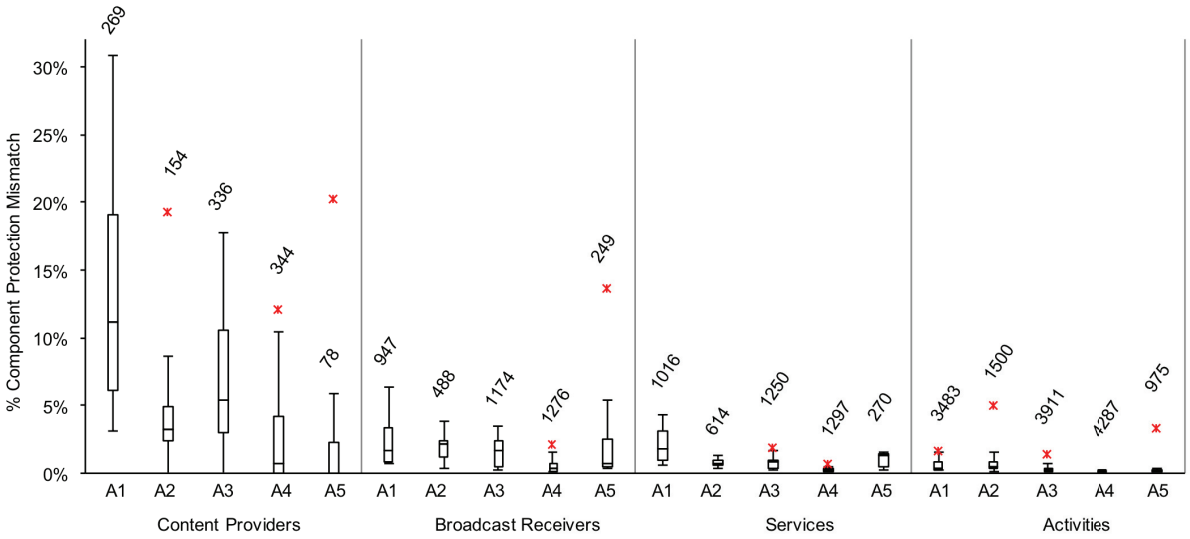


Figure 8: Components Protection Mismatch Breakdown