International Journal of Teknology Informatics and Engineering
e-ISSN :2961-8215;.p-ISSN :2961-9068
Volume 1 No. 3 , Desember - 2022

CREDENTIAL ANALYSIS FOR SECURITY CONFIGURATION ON CUSTOM ANDROID ROM

Joseph Teguh Santoso
Universitas Sains dan Teknologi Komputer

Fujiama Diapoldo Silalahi
Universitas Sains dan Teknologi Komputer

Laksamana Rajendra Haidar
Universitas Sains dan Teknologi Komputer

Abstract. Android is an operating system with open source and consists of several layers, with the different layers its duties and responsibilities. Various parties in the customization chain such as device vendors such as Samsung, Xiaomi, Oppo, Huawei, and others, operators such as Telkomsel, Smartfren, XL, etc., and hardware manufacturers can customize one or more layers to adapt devices for different purposes, such as supporting specific hardware and providing different interfaces and services.
The purpose of this study was to investigate systematically for any inconsistencies that arose as a result of the processes involved in this study and to assess their various security implications. This research runs DroidDiff to perform a substantial-balance diverse investigation on images collected by the analytical methodology. DroidDiff found a lot of differences when it comes to the selected features. The method used in this study is the method of five differential analysis algorithms. As a result, by comparing the security configurations of similar figures, important security changes that could be accidentally introduced during customization can be found.
The results show that DroidDi can be used by vendors to check the configuration of various security features in a given image. DroidDiff will extract those features from the image, and compare them to other image configuration sets, then DroidDiff will flag the inconsistent ones for further investigation by vendors who have the source code and tools to check their effect. For future work, improvements to DroidDi to more accurately detect risky inconsistencies are highly recommended. Improving DroidDiff will help reduce the number of false positives and determine risky configurations more accurately.

Keywords: Credential Analysis, Android, Security Configuration, Android Customization.

PRELIMINARY

Like the leaves that bloom in the rainy season, the smartphone market is growing rapidly for various reasons. Smartphone with Android OS has such popularity in various circles without age limits. Android drives innovation and brings more and more features to its users, this makes it a feature-rich device and an attractive mobile application. The open nature of the Android ecosystem adopted by Google naturally provides the basis for a highly fragmented operating system. The official version of Android is enterprisingly custom-made into a multitude of organization images by people in the customization string with hardware builders, material vendors, and carriers free to create the basis for customizing it with various features and models to differentiate their products from competitors. As device vendors become more adept at tweaking the Android framework and default system apps, making features like higher resolution and more innovative cameras eventually make Android customization more prevalent over time. However, these customizations lead to problems, present serious security vulnerabilities, and ultimately cause severe damage. Given that, and the security of the vast consumer Android customizations at stake, it's important to study and investigate the security ramifications of Android.

Android Customization Process

The utilization zone, placed at the dominant place of the Android architectural model and providing core applications such as Home, Contacts, Phone, and Browser, is often customized by device vendors and carriers. Research by Jiang et al., (2013) has shown that this layer has always been the focus of vendor customization. Vendors and operators modify AO-SP applications that are reloaded by default to provide more advanced functionality and UI. The Framework and Libraries layer sits immediately below the Application layer and provides support for developers to access a variety of privileged resources, services, and functionality. At the base of the layer is the "Linux kernel", providing a level of separation among the instrument hardware and containing all the important hardware drivers.

Android Customization Dangers

Android device customization provides opportunities for enhanced functionality and excellent personalization for its customers, however, it also provides opportunities for increased security risks, given that Android's fragmentation process is highly unregulated. To keep this nether supervision, Google has launched an "Android Compatibility Program" to guide the personalization process. At the Linux kernel layer, a study by Wang, (2014) investigated "the security configuration of Android Linux device drivers and found that many of these devices have not been properly protected, causing their exposure to parties who should not access them can directly command the open camera driver to take a picture". Research by Jiang et al., (2013) revealed that the Application layer is also full of flaws introduced during the customization process. Similarly, the work of Jiang et al., (2012) anecdotally shows that Android vendor-preloaded apps have security flaws delivered on some special devices.

LITERATURE REVIEW

Android Customization - Vendors customize the Android framework to support more advanced features and offer a unique user experience. Complicating the process further, the baseline of AO-SP has been heavily modified for different versions of Android. Of the 682,000 devices surveyed, more than 24,000 different devices and 1294 different brands have been identified, of which the largest portion (37.8%) is manufactured by Samsung. **Specific Hardware Components -** Over time, Android devices may run different hardware components. Hardware and chipset manufacturers customize Android devices to provide better levels of performance and more advanced hardware. Vendors adopt the primary Android software bundle to new hawker-limited hardware platforms by integrating some of the add-ons to the Android OS. **Interface Device Vendor Specific Users -** Some vendors design their UI, and each custom UI uses its color palette and UI elements, e.g. different UI on Samsung and Xiaomi Redmi, each presenting a different setup aesthetically. Device vendors integrate this custom UI into their own devices by adding and modifying existing Stock Android UI elements. **Carrier-**

Specific Features - Based on carrier network requirements (Tsel, XL, 3, and so on), device vendors modify telephony services to allow integration of multiple LTE and GSM bands, even 5G networks that are currently released. Device vendors must also make a series of Android layer changes, to support carrier-specific restrictions. **Android Update - Android** customization process at a fast pace AO-SP updates its OS versions, most of which are already highly customized, and gives rise to the multitude of personalized Android section synchronic on lots of cell phones worldwide at various version levels and coexisting today's market.

Customization Effects

Compatibility and Portability Issues

Each new Android API level introduces features and removes bugs. Even if each release aims to integrate new changes without breaking existing pre-installed applications in older versions, often perfect compatibility cannot be achieved. Due to insufficient product lines and cross-platform testing for new features and changes, Android apps may not behave consistently across versions. To ensure consistent and correct Android application behavior across various API levels and hardware configurations, Google presents the "Firebase Test Lab" (FTL) for Android, a cloud-based infrastructure for comprehensive testing of Android applications before they are released, through FTL Android developers can access various devices. Android devices installed in Google's data centers and testing their apps across different makes and models, across different Android APIs, device configurations, and screen orientations, and to locate dissolution and establish unity over the Android devices, Google launched the Compatibility Program earlier.

Android Layered Architecture

To enable the application builder to access different assets and functionality, the Android scheme layer brings many high-level of initiation equally PackageManager, ActivityManager, NotificationManager, and many others. This service mediates access to system resources and enforces appropriate access oversight based on several criteria such as application user id, Android permissions obtained, Signature, etc. Just down the structure layer is the "Libraries layer" which is a collection of "Android-specific libraries" and other core libraries like "libc, SQLite database, media libraries", etc. Just like the structure initiation, convinced Android custom libraries to perform multiple access control analyses based on similar criteria (e.g. caller user id, permissions, etc). At the bottom of the Android layer is the Linux kernel which provides a level of abstraction between the device hardware and the layers over it. It consists of all the small-level of foundation system functionality like recollection management, action, and energy supervision and brings essential hardware drivers like a camera, a display device, and Wifi. The "Linux kernel layer" intercedes access to hardware drivers and rough assets based on the basic "Discretionary Access Control (DAC)".

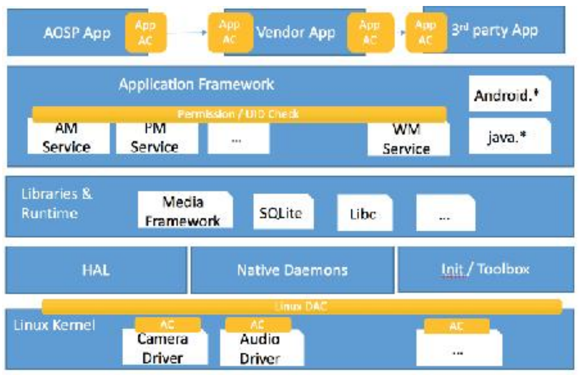


Figure 1.: Android Architecture and Layered Components

Customization Aspect: Application Layer

- ***Added new default application.*** Wu et al (2013) conducted a source analysis that aimed to characterize different pre-prosperous applications into 3 clusters: applications that came from AO-SP, applications personalized by vendors, and other applications that were merged into ordinary images with at most 18 results, 22% of apps are from AO-SP, implying that 81.78% of preloaded apps are added by vendors and others.
- ***Modifies existing AO-SP and older vendor-specific app editions.*** During the customization process, vendors may add new components to existing preloaded applications to provide new functionality and services. To adapt applications to different needs, vendors may also change the implementation of one or more components.
- ***Removing existing preloaded applications***. Vendors may customize Android devices by removing certain applications that are not required for the functionality of the device or that may have been replaced with vendor-specific applications.

Customization Aspect: Framework Layers

- Added new system services
- Modify existing system services
- Modify other framework binaries
- Changing configuration files

Customizable Security Risk Categorization

1. ***Security Risk due to the Addition***

Figure 2 summarizes the possible security effects of introducing new applications, components, or services to kernel frameworks, libraries, and drivers.

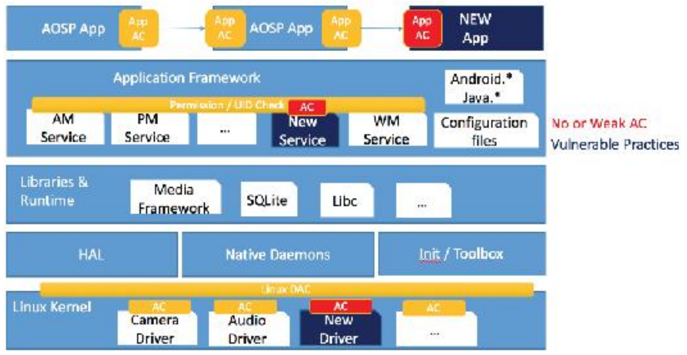


Figure 2.: Customization Security hazard from adding a new entity

- ***Buggy and Accessible Network Application***. With no attentively calculated and supplied, newly introduced applications or built-in entrails might consist of harmful practices, research by Jiang et al., (2013) shows that most of the vendor-preloaded apps studied exhibit permissions that exceed privileges. "This study detects that most of the vendor applications contain static willing effluence and content pollution" (Zhou&Jiang, (2013)). Content leaks occur when the content provider is world-readable, or if it is accessible from other exposed components.
- ***Vulnerable framework services and libraries***. Omitting access control checks in vendor-added system service APIs could expose related functionality to unauthorized applications, similarly, weaker access control checks could also compromise associated operations, as they can be easily circumvented.
- ***Weak new device driver configuration.*** If not properly protected, devices could grant illegitimate apps to access hypersensitive user data or system capabilities which usually require malicious/Android system permissions Therefore, vendors can harm devices if they do not enforce proper access control permissions in device drivers recently added.

2. **Security Risk due to Modification**

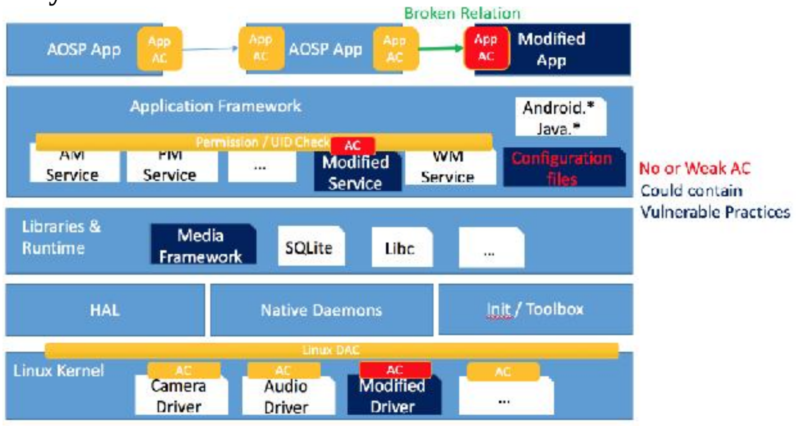


Figure 3.: Customization Security hazard from modifying existing entities

- **Vulnerable System Apps.** If the elemental components provide powerful prospects, humiliating their conservation will consistently advantage to accepted Android susceptibility equally permissions representation offensive, gratified exposures, and infection invasion.
- **Powerless system-expanded structure**. If not done carefully, the new security configuration could be weak and break some of the assumptions made by other components. If you enter the wrong G-ID into the permissions mapping then the permissions level will be lowered, so it is very dangerous to continue.
- **Weaker configuration of existing device drivers**. Despite analyzing the device driver configurations of 2423 images and comparing them to their counterparts in AO-SP, the study by Wang et al., (2014) identified 1290 possibly vulnerable images, including at least one device driver whose write/read access is weaker than the same file in AO-SP reference.
- **Disconnects in Android parts.** Various Android parts are linked simultaneously by "Inter-Component Communication (ICC)". By modifying component identifiers and intent filter attributes during customization, vendors risk solving intrinsically existing ICC between various preloaded applications, installed third-party

applications, and even framework services that refer to other components in the device.

3. **Security Risk due to Deletion**

Removing certain applications or components from existing applications can cause a break in the real connection that stands between them. If used on a device but its specifier has been detached, a malware application could be enough in those gaps to gain important systems potentially, simply by impersonating the owner of the attribute.

RELATED WORK

Security Dangers of Android Customization - Research by Jiang et al., (2012) who systematically studied 8 popular Android smartphones from various manufacturers revealed that these phone images did not accurately accomplish an admission-based safety model. Some special or critical acceptance that covers perceptive user data in preloaded apps will be exposed to non-privileged apps. Research by Wang et al., (2014) analyzes the security configuration of Linux Android device drivers to find the dangers of fragmentation at the Kernel layer, Wang et al., (2014) conducts efficient research on the safety risk of customizing Android devices through automatic identification of associated Linux files by operating on a specific device driver then comparing the level of protection (Linux file permission bits for each file) on the vendor version with the appropriate AO-SP version. Any weak protection detected in vendor device drivers implies a potential security hazard. The existence of a Linux file permissions mismatch across two similar OSes, together with its dangerous relationship with Android permissions is quite worrying. Tian et al., (2014) audited third-party Android phones oblique with the authoritative Android operating system to find potential security vulnerabilities and design flaws that glide over the hawker personalized. Tian et al., (2014) extracted preinstalled apps and libraries from a custom Android image, built a suitable system from AO-SP, then compared preinstalled apps and libraries to find modifications and assess their safety. Research by Gallo et al (2015), highlights security issues in the Android permission model concerning Android customization.

Improved Android Security. Wallach et al., (2011) proposed a security mechanism to overcome the problem of deputy confusion by tracing the IPC call chain and allowing selected applications to operate with reduced caller privileges or exercise their full privileges. Similarly, research by Wetherall et al., (2011) introduces new privacy controls to protect sensitive user data by providing shadow data instead of personal data, and through exfiltration blocking. Other frameworks such as XManDroid (Sadeghi, (2011)) and "TrustDroid" (Shastry et. al., (2011)) "focus on mediating communication between components in different applications". "FlaskDroid" (Sadeghi et al., (2013)) and the SEAndroid project (Craig et al., (2013) also mediate component interactions as part of their deployment. SEAndroid solves the professionally complicated challenge of "porting SELinux-based essential access control" from the desktop territory to Android.

Android update. Some researchers are trying to identify vulnerabilities caused by the very fast Android application life cycle and frequent updates released by Google. Thomas et al., (2015) compiled a corpus of 20400 dedicated Android devices and demonstrated that is important changeability in the delivery of safety updates to Android devices manufactured by different vendors and carriers; which leads to a known unpatched security vulnerability. The research revealed that 87.7% of Android devices pooled had major safety susceptibility and were resolved to 1 big hazard. Research work by Yuan et al., (2014)) reveals another class of attacks caused during an Android OS update where an attacker can strategically invoke other permissions and attributes, which are available in future versions of the OS, to elevate his privileges after the update is done.

Component Hijacking Attack on Android.

The security effects of exporting content providers have been studied by Zhou et al., (2013) including content leakage and content pollution. Permission re-delegation attacks (Chin et al., (2011)) illustrate another consequence of unintentional export of public interfaces, in which an application with permissions performs privileged tasks on behalf of the application without that permission; thus, attackers can use privileged capabilities without obtaining the appropriate Android permissions.

Android malware

Potharaju et al., (2012) proposed various hazard indicators based on the permissions needed, for the division, such as the acceptance requested from applications belonging to the same division. Molloy et al.'s research, (2012) used a probabilistic generative model to calculate the real hazard mark of Android applications based on the acceptance they demand. Other work to detect malware via bytecode level information includes which relied on an exploratory and failure way to analyze skeptical arrangement in expert code, and "DroidRanger" (Jiang et al., (2012)) which detects Android malicious software based on the similarity of needed acceptance and behavior traces to known malicious software families, formulated through heuristic based filtering. DroidAPIMiner performs thorough API call frequency analysis in both gentle and malware applications to extract malware appearance and adopt machine learning to get the largest compatible appearance.

AsDroid Liang et al., (2014) "detect hidden application behavior by identifying discrepancies between API calls and text displayed in the GUI". Rieck et al., (2014) "extracted several features from Android applications and applied machine-learning techniques to perform the classification". Zhao et al., (2014) "extract more sophisticated classification features to counter malware variants and zero-day malware". More specifically, they extract contextually weighted API addiction graphs as programmatic definitions to build up appearance sets and introduce chart affinity metrics to discover comparable utilization action suiting indulgent secondary application characteristics.

Methodology

Feature Extraction

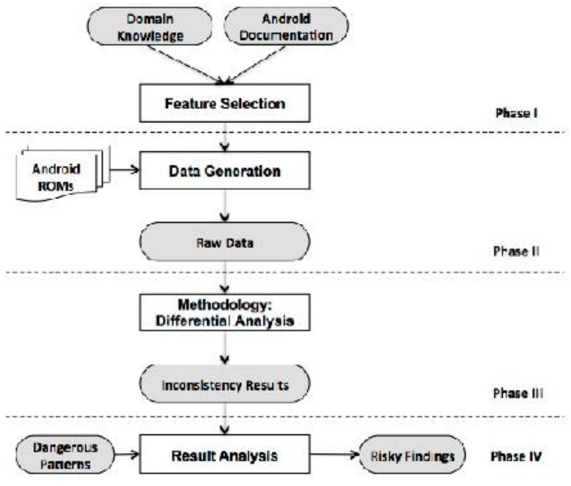


Figure 4: Investigation Flow

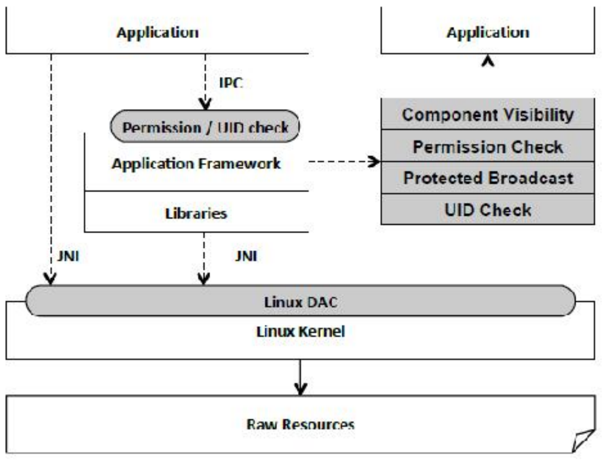


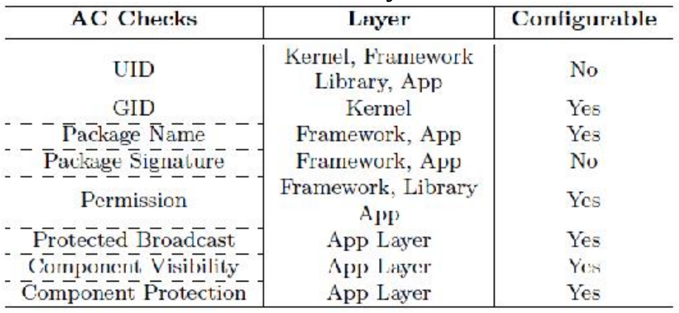
Figure 5: Android Security Model

1st dominant layer is preloaded applications given by instrument hawkers and another 3 parts (mobile carriers). To enable the utilization builder to access a variety of assets and functionality, the "Android Framework" layer brings many high-level services. This service mediates access to system assets and enforces appropriate access control based on the application's user id and the Android acceptance it has obtained. Assertive services may accomplish access control based on the package name or caller credential. Just down the structure layer is the "Libraries layer" a set of "Android-specific libraries" and other crucial libraries. Just the scheme services, convinced Android custom libraries to perform various access control analyses based on the caller's user id and acceptance as well. The bottom of the layer is the "Linux kernel" that brings a level of absorption among the instrument hardware and consists of all the important hardware drivers. The "Linux kernel layer" negotiated access to hardware drivers and raw assets based on the standard "Discretionary Access Control (DAC)". To boost association, Android applications are linked gather by "Inter-Component Communication (ICC)". Apps can invoke components of other utilization (such as activities and services) through intent mechanisms. It could configure some safety parameters to cover the asset's band functionality.

Permission

Default and custom Android permissions are used to protect inner components, data, and functionality. The level of permission protection must be chosen carefully depending on the resources to be protected. Signature and SystemOrSignature level permissions are used to protect the most privileged resource and will only be granted to applications signed with the same certificate as the specified application. Dangerous permissions protect personal data and resources or operations that affect data stored by the user or other applications such as reading contacts or sending SMS messages. Requesting Danger level permissions requires explicit user confirmation before granting them. The normal level, on the other hand, is granted permissions that protect the least privileged resource and do not require user approval.

Table 1: Security Check



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

Regularly, for different describe acceptance e ∈ ep defines the safety appearance fine as follows:

$$fine = ProtectionLevel(e)$$

The possible value of one is in the set "{Normal, Dangerous, Signature, Unspecified, 0}". The SignatureOrSystem level is mapped to Signature because neither of them can be obtained by third-party applications without a Signature check. Undefined values refer to predefined permissions without protection level, whereas 0 refers to undefined permissions in the image.

GID

Certain low-level Linux ID groups (GIDs) map to Android permissions. Once an application process has obtained permissions then the G-ID is mapped and used for access control in the kernel. Android maintains that any inadvertent changes to platform.xml will expose a serious vulnerability. To allow the discovery of GIDs that are vulnerable to permission mapping, the minimum permission requirements required to obtain a particular G-ID are mapped to the given image. If the same G-ID has each minimum requirement on 2 images, this is probably accessible.

Protected Broadcast

Safety broadcasts are broadcasts that can only be sent by system-level processes. Applications use protected broadcasts to ensure that no system-level process can trigger a particular broadcast receiver. System applications can define broadcasts that are protected as follows:

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

Other applications can use the protected broadcasts described above via the following:

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

The above receivers can only be generated by the system process that broadcasts a broadcast. name-protected broadcast. The application can also use protected broadcasts via dynamically registered broadcast receivers. While the personalization process, certain bundles are updated and changed. Certain protected broadcast definitions will be removed as well. The goal is to uncover whether these conflicting unprotected broadcasts are still being used, as the process of filters inside the receiver. This may open up a deliberate vulnerability, as receivers that developers thought could only be called by system processes would now be callable by any third-party application and consequently expose their functionality. Properly, for different "Protected Broadcast $e \in E_{PB}$" is defined as:

$$fn_e = DefineU \quad (e),$$

Part of Clarity

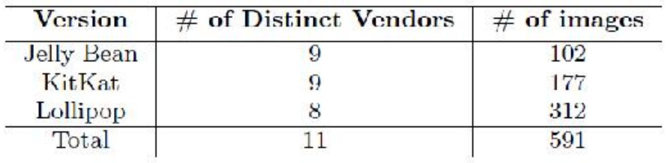
Android grants the builder the to determine if only declared parts could be termed apparently by another application. Clarity could be rooted via flags exported in part declarations inside the app's manifest file. If this flag is unspecified, clarity will be essentially rooted based on whether the part describes a filter when present then the part will be exported otherwise it will appear as:

```
// 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>
```

Data Creation

To reveal whether the customization party changed the configuration of the quoted safety appearance, a bigger-scale divergent investigation was performed. a total of 591 Android ROMs were collected from Samsung Update and physical devices. These images were personalized by 11 hawkers (vendors), for approximately hundred and thirty-five models, and eight carriers. They operate Android versions from 4.1.1 to 5.1.1. Details about the collected images are in Table 2. In total, this image includes an average of 157 apps per image and 93169 of all apps simultaneously. To extract the selected security feature values in each image, a tool named DroidDi was developed in this study. For different images, “DroidDi ” first takes its scheme resource “Apks” and preloaded Apks, then runs “Apktool” to extract the analogous manifest files. Second, it collects configuration files under “/etc/permission/”. Then, “DroidDiff” searches the extracted manifest and configuration files for the definitions of the targeted entities (EP, EPB, EGID, and EC). Lastly, “DroidDiff” runs the values developed over the various investigation methodology.

Table 2: Android Image Set



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

RESULTS AND FINDINGS

This research runs DroidDiff to perform large-scale differential analysis on images collected by adopting the preceding method. DroidDiff found a lot of differences when it comes to the selected features. Figure 6 shows the global development identified by the investigation of this study. The moderate proportion of identified inconsistencies is plotted for various appearance parts (Permissions, GID, Broadcast Protected, Component Visibility, and Component Protection) using five differential investigation algorithms. To bring an opinion of the amount of inconsistency, different grid trick represents the average amount of total global existence in the studied image set. Using the first grid of tiles as a specimen to draw what the data means: under the “Cross-Version (A1)” investigation, “DroidDiff” developed an average of 673 global acceptance per different studied applicant set. Fifty percent of the applicant image pool contains a partial four points eight percent of the amount of acceptance that has conflicting levels of protection. Figure 6 characterizes the root of images that are deviation, they have a bigger amount of differences correlated to another root of images in the same array.

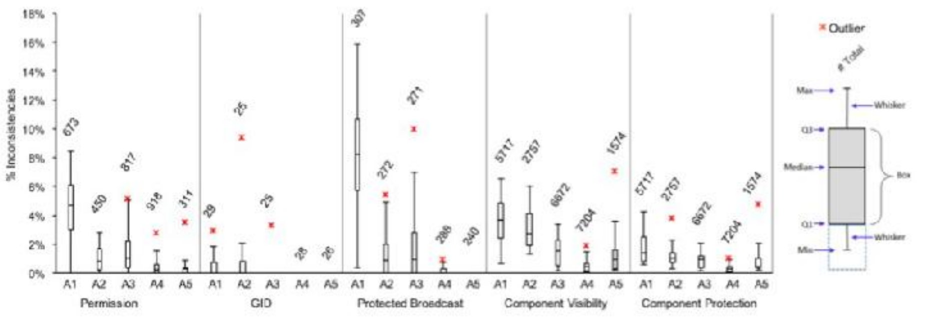


Figure 6: Overall Inconsistency Detected. Al: Cross Version, A2: Cross Vendor, A3: Cross Model, A4: Cross Carrier, US: Cross Region

As described in Figure 6, the "Cross-Version (A1)" investigation catch the maximum proportion of disagreement across all five division, that's means, improvement of the identic device model to various Operation System version propose the best safety configuration improvement. The perceptive explanation trailing this over the new Operation System clemency, Android may apply greater conservation to the identical individual to mass several bugs that found (example: expanding permission requirements to privileged services). However, through more recent OS releases, it was discovered certain security features were downgraded, leading to potential risks if done accidentally.

Through the "Cross-Vendor (A2)" investigation, "DroidDiff" detected that some safety appearances were incompatible between hawkers, like if had an identic Operation System version. Further analysis was performed on the vendor causing the top number of deviations. A compelling observation is that lower Hawker, as well as Xiaomi, introduce some hazardous incompatible. All those incompatible;e (potentially egregious) G-IDs are actually due to these 3 companies. Perhaps, small vendors may not have sufficient expertise to fully evaluate the security implications of their actions. Cross-Model Analysis (A3) also detected several inconsistencies, meaning that different device roles by the same hawker and Operation System version may have various safety configurations. Even though the "Cross-Carrier (A4)" and "Cross-Region (A5)" investigations catch a low proportion of inconsistencies, it is closely important to notice the identical device model running the same Operation System version may have several various configurations if adjusted for various devices operator.

Permission Change Arrangement

Safety level Imbalance

"DroidDiff's" differential analysis outcome clarifies that Android acceptance might have various levels of protection on identical images. As illustrated by Figure 6, more than 50% of candidate image pools contain at least 32 (of 673), and 9 (of 817) permissions that have inconsistent levels of protection across Cross-Version (A1) and Cross-Model (A3) analyzes, each. To reveal more insights, the most common protection level change combinations are also examined, which of the following three desirable combinations is highly frequent "(Normal, Dangerous), (Normal, Signature), or (Dangerous, Signature)" The occurrence of each pattern is also calculated, and the outcomes are shown in Figure 7. The combination "(Normal, Signature)" is a highly frequent arrangement. This is over genuine because some acceptance has a "Signature protection level" on several images are appointed by a "Normal protection level" on other images. Presented here are 2 acceptance that has incompatible levels of protection:

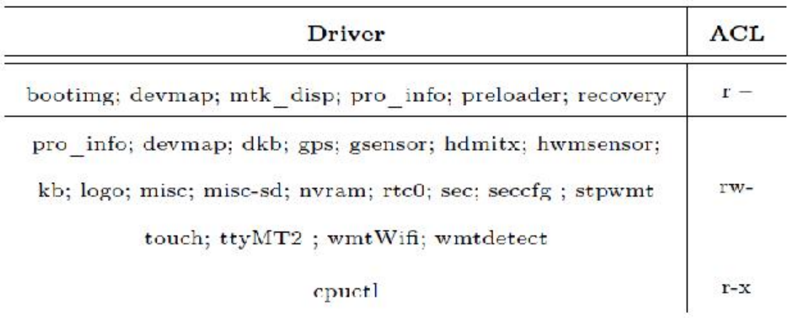
- com.orange.permission.SIMCARD_AUTHENTICATION holds Signature and Normal protection on Samsung S4(4.2.2) and Sony Experia C2105 (5.0.1), respectively.
- com.sec.android.app. syncope.permission.RUN_ SYSSCOPE holds Dangerous and Signature protection on Samsung Note4 and S4(5.0.1), respectively.

Attack

Several actual attacks have been confirmed on several devices, among others.

1. **Steal e-mails**. SecEmailSync.apk is the default application on Samsung devices. This consist of a satisfied operator "com.Samsung.android.email. other providers". Over-role analysis reveals inconsistent permission protection at this operator around multiple Samsung images. “Read and Write” access to this operator is guaranteed with the Signature acceptance “com. Samsung.android.email.permission.”
2. **Fake a Prime SMS message**. The “TeleService Package (com.android.phone)” comes pre-loaded on many Samsung devices and brings a lot of outset for phone and call authority. An essential service is “the.TphoneService”, which shows several key telephony functions such as receiving voice and video calls, dialing a new phone number, mailing letters, and recording voice and video calls. “Cross-Model” and “Cross-Version” analysis reveals permissions mismatches in this demanding authority
3. **Invalid “factory reset”**. Announcement receiving ServiceModeAppBroadcastReceiver that listens for multiple intent filters including the action filter com. Samsung.intent.action.SEC_FACTORY_RESET_ WITHOUT_FACTORY_UI which allows resetting the phone and deleting all data without user confirmation. Cross-Version analysis reveals a mismatch of protection for this important broadcast receiver. On most devices running Kitkat and below, this receiver is protected with the Signature permission com.sec.android.app.servicemodeapp.permission.KEYSTRING. However, on some Lollipop images, it is not properly protected.
4. **Access essential drivers with normal permissions**. Over-Hawker investigation declares demanding protection downgrades of system GIDs. On some images, such as the Samsung S5 (4.4.2), this G-ID maps to the Signature permission com.qualcomm.permission.IZAT. However, in other images, this G-ID maps to the normal level of the android.permission permissions. ACCESS_MTK_MMHW indicates that any third-party application can easily get the system GID. Table 4 lists the device drivers that can be accessed via the system G-ID on the Digiland DL700D Tablet. This is a privileged driver but is now accessible to normal applications.

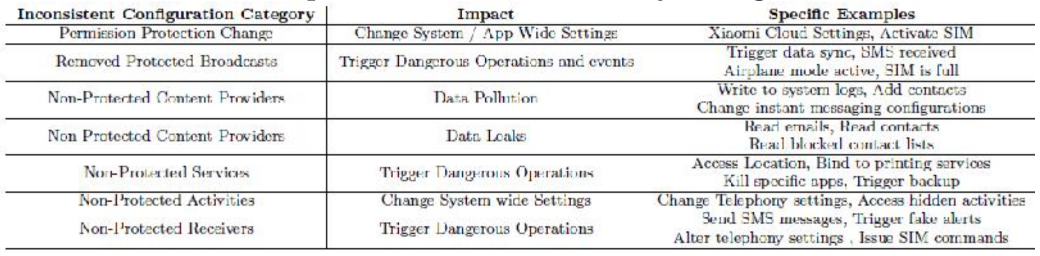
Table 3: Drivers accessible by the System GID



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

5. **Triggering an emergency broadcast without permission**. CellBroadcastReceiver is a built-in Google app that performs important functions based on the cellular announcement it receives. These annals a PrivilegedCellBroadcastReceiver broadcast receiver that allows it to receive emergency broadcasts from the cell provider (eg evacuation warning, presidential alert, yellow alert, etc.) and displays the appropriate alert. This important function could bring about the “android.provider.Telephony.SMS_EMERGENCY_CB_RECEIVED” action being received. “Cross-Vendor” and “Cross-Version” analysis found preservation incompatibility in this acceptor with different devices but protected with android.permission Dangerous “permission.READ_PHONE_STATE”. analysis revealed that this was also caused by a risky pattern of duplicate recipients. On the victim’s device, “PrivilegedCellBroadcastReceiver” has been announced 2x, 1st announcement needs the “Signature permission” and stem the android.provider.Telephony.SMS_EMERGENCY_CB _RECEIVED action, while the second takes care of a less privileged action and requires Dangerous permission. Any third-party application can bypass permission requirements on the first recipient via explicit invocation.
6. **Corrupts “system-wide settings”. “**SystemUI” is a built-in application that commands system windows. It handles and draws much of the system UI such as the top status bar, system notifications, and dialogs. To manage the top status bar, Samsung SystemUI specifically includes the com.android. system.PhoneSettingService service, which handles incoming requests to turn on/off various system-wide settings that appear on the top status bar. These settings include turning on/off wifi, Bluetooth, location, mobile data, tathering, driving mode, etc; which is usually done with the consent of the user. The analysis performed shows the incompatibility of protections for these services. On S5(4.4.2) and Note8(4.4.2), this service is guaranteed by the Signature permission “com.sec.phonesettingservice.permission.PHONE_ SETTING”.
7. **Another “Randomly Selected Cases”**. The effect of inconsistent safety configurations is important. In addition to the end-to-end attack, a total of 40 inconsistencies were randomly selected and manually analyzed for what could have happened after being exploited.

Table 4: 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

Limitations

- **Component implementation changes.** Static changes to a component's security configuration (visibility or protection permissions) may not necessarily represent a security risk over time.
- **Component renaming.** The approach used will miss detecting inconsistent component configurations that have been renamed during customization.

CONCLUSIONS AND RECOMMENDATIONS

This research conducts a study of Android customization related to security aspects. This study aimed to systematically investigate any inconsistencies created as a result of this process and to assess their security implications. The first investigation led to the discovery of a genuine Android safety defect that had never been considered previously. This research highlights the importance of security risks and reveals crashes in various Android devices, and shows that Android devices are full of such flaws. Second, this research is making the early experiment to consistently find safety configuration adjustments imported by Android personalization. This research lists the safety appearance implemented across different layers of Android and utilizes differential analysis in a substantial number of personalized ROMs to catch out if persistent across those layers. By measuring the safety configurations of identical drawings, important security changes that could be accidentally introduced during customization can be found. The results show that DroidDi can be used by vendors to check the configuration of various security features in a given image. DroidDiff will extract those features from the image, and compare them to other image configuration sets, then DroidDiff will flag the inconsistent ones for further investigation by vendors who have the source code and tools to check their effect.

For future work, this research suggests improving DroidDi to more accurately detect risky inconsistencies. In addition, similarities between components must also be calculated and their security configurations must be examined more thoroughly to detect cases that might have been missed. Improving DroidDiff will help reduce the number of false positives and determine risky configurations more accurately. So that DroidDifferential analysis results can be used to predict the correct security configuration of misconfigured features. If most of the security features have the same configuration, then the inconsistent components will likely be configured the same way in the victim image.

BIBLIOGRAPHY

A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “Andromaly: a Behavioral Malware Detection Framework for Android Devices,” Journal of Intelligent Information Systems archive Volume 38 Issue 1, 2012.

AP Felt, HJ Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission re-delegation: attacks and defenses,” in Proceedings of the 20th USENIX conference on Security symposium, 2011.

B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, “Android Permissions: A Perspective Combining Risks and Benefits,” SACMAT, 2012.

D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, “Drebin: Effective and explainable detection of android malware in your pocket.,” in NDSS, The Internet Society, 2014.

D. Feth and C. Jung, Context-Aware, Data-Driven Policy Enforcement for Smart Mobile Devices in Business Environments, pp. 69–80. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

DR Thomas, AR Beresford, and A. Rice, “Security metrics for the android ecosystem,” in Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '15, (New York, NY, USA), pp. 87–98, ACM, 2015.

G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich, “Yaase: Yet another android security extension,” in Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third International Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on, pp. 1033–1040, 2011.

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 2012 ACM conference on Computer and communications security, 2012.

I. Burguera, U. Zurutuza, and S. Nadijm-Tehrani, “Crowdroid: Behavior-Based Malware Detection System for Android.,” SPSM, 2011.

J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, “Asdroid: Detecting stealthy behaviors in android applications by the user interface and program behavior contradiction,” in Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, (New York, NY, USA), pp. 1036–1046, ACM, 2014.

K. Tam, SJ Khan, A. Fattori, and L. Cavallaro, “Copperdroid: Automatic reconstruction of android malware behaviors.,” in NDSS, The Internet Society, 2015.

KZ Chen, NM Johnson, V. D'Silva, S. Dai, K. MacNamara, TR Magrino, EX Wu, M. Rinard, and DX Song, “Contextual policy enforcement in android applications with permission event graphs.,” in NDSS, The Internet Society, 2013.

L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, “The impact of vendor customizations on android security,” in Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS '13, (New York, NY, USA), pp. 623–634, ACM, 2013.

L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, “Upgrading your android, elevating my malware: Privilege escalation through mobile os updating,” in Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP '14, (Washington, DC, USA), pp. 393–408, IEEE Computer Society, 2014.

LK Yan and H. Yin, “Droidscope: seamlessly reconstructing the os and Dalvik semantic views for dynamic android malware analysis,” in Proceedings of the 21st USENIX conference on Security symposium, 2012.

M. Lindorfer, M. Neugschw, L. Weichselbaum, Y. Fratantonio, VVD Veen, and C. Platzer, “Andrubis- 1,000,000 apps later: A view on current android malware behaviors,” in International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, 2014.

M. Mitchell, G. Tian, and Z. Wang, “Systematic audit of third-party android phones,” in Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY '14, (New York, NY, USA), pp. 175–186, ACM, 2014.

M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Ho mann, “Mobile-sandbox: Having a deeper look into android applications,” in Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, (New York, NY, USA), pp. 1808–1815, ACM, 2013.

M. Zhang and H. Yin, “Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications,” in NDSS, 2014.

M. Zhang and H. Yin, “E cient, context-aware privacy leakage confinement for android applications without firmware modding,” in Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, (New York, NY, USA), pp. 259–270, ACM, 2014.

M. Zhang, Y. Duan, H. Yin, and Z. Zhao, “Semantics-aware android malware classiﬁcation using weighted contextual API dependency graphs,” in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, (New York, NY, USA), pp. 1105–1116, ACM, 2014.

MC Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic detection of capability leaks in stock android smartphones,” in 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5 -8, 2012, 2012.

MI Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, “Information-flow analysis of android applications in droidsafe,” 2015.

MY Wong and D. Lie, “Intellidroid: A targeted input generator for the dynamic analysis of android malware,” in NDSS, 2016.

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, (New York, NY, USA), pp. 639–652, ACM, 2011.

P. Pearce, AP 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, 2012.

R. Gallo, P. Hongo, R. Dahab, LC Navarro, H. Kawakami, K. Galvão, G. Junqueira, and L. Ribeiro, “Security and system architecture: Comparison of android customizations,” in Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec '15,(New York, NY, USA), pp. 12:1–12:6, ACM, 2015.

S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, (New York, NY, USA), pp. 259–269, ACM, 2014.

S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A. -R. Sadeghi, and B. Shastry, “Practical and lightweight domain isolation on android,” in Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, 2011.

S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. -R. Sadeghi, “Xmandroid: A new android evolution to mitigate privilege escalation attacks,” tech. rep., Technische UniversitÃÂČÂâĆňt Darmstadt, 2011.

S. Bugiel, S. Heuser, and A. -R. Sadeghi, “Flexible and fine-grained mandatory access control on android for diverse security and privacy policies,” in Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), (Washington, DC), pp. 131–146, USENIX, 2013.

S. Jana and V. Shmatikov, “Memento: Learning secrets from process footprints,” in Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, (Washington, DC, USA), pp. 143–157, IEEE Computer Society, 2012.

S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, “Harvesting runtime values in android applications that feature anti-analysis techniques,” in NDSS, 2016.

S. Shekhar, M. Dietz, and DS Wallach, “Adsplit: Separating smartphone advertising from applications,” in Proceedings of the 21st USENIX Conference on Security Symposium, Security'12, (Berkeley, CA, USA), pp. 28–28, USENIX Association, 2012.

S. Smalley and R. Craig, “Security enhanced (SE) Android: Bringing Flexible MAC to android,” in 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013, 2013.

W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. -G. Chun, LP Cox, J. Jung, P. McDaniel, and AN Sheth, “Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones,” ACM Trans. Comput. Syst., vol.32, pp. 5:1–5:29, June 2014.

W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, “App context: Differentiating malicious and benign mobile app behaviors using context,” in Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, (Piscataway, NJ, USA), pp. 303–313, IEEE Press, 2015.

X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, CA Gunter, and M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and DS Wallach, “Quire: Lightweight provenance for smartphone operating systems,” in 20th USENIX Security Symposium, (San Francisco, CA), Aug. 2011.

X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, “The peril of fragmentation: Security hazards in android device driver customizations,” in 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA.

Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, G. Vigna, S. Uc, and Barbara, “Triggerscope: Towards detecting logic bombs in android applications,” in S&P, 2016.

Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, XS Wang, and B. Zang, “Vetting undesirable behaviors in android apps with permission use analysis,” in CCS, (New York, NY, USA), ACM, 2013.

Y. Zhou and X. Jiang, “Detecting passive content leaks and pollution in android applications,” in 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013,2013.

Y. Zhou, X. Zhang, X. Jiang, and VW Freeh, “Taming information-stealing smartphone applications (on android),” in Proceedings of the 4th International Conference on

Trust and Trustworthy Computing, TRUST'11, (Berlin, Heidelberg), pp. 93–107, Springer-Verlag, 2011.

Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, You, Get off of My Market: Detecting Malicious Apps in O cial and Alternative Android Markets,” NDSS, 2012.

Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, XS Wang, Z. Qian, and H. Chen, “revdroid: Code analysis of the side effects after dynamic permission revocation of android apps,” in Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS '16, (New York, NY, USA), pp. 747–758, ACM, 2016.