August 2016

SYSTEMATIC DISCOVERY OF ANDROID CUSTOMIZATION HAZARDS

Yousra Aafer
Syracuse University

ABSTRACT

The open nature of Android ecosystem has naturally laid the foundation for a highly fragmented operating system. In fact, the official AOSP versions have been aggressively customized into thousands of system images by everyone in the customization chain, such as device manufacturers, vendors, carriers, etc. If not well thought-out, the customization process could result in serious security problems. This dissertation performs a systematic investigation of Android customization' inconsistencies with regards to security aspects at various Android layers.

It brings to light new vulnerabilities, never investigated before, caused by the under-regulated and complex Android customization. It first describes a novel vulnerability Hare and proves that it is security critical and extensive affecting devices from major vendors. A new tool is proposed to detect the Hare problem and to protect affected devices. This dissertation further discovers security configuration changes through a systematic differential analysis among custom devices from different vendors and demonstrates that they could lead to severe vulnerabilities if introduced unintentionally.

SYSTEMATIC DISCOVERY OF ANDROID CUSTOMIZATION HAZARDS

by

Yousra, Aafer

M.S., Syracuse University, December 2011

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Syracuse University
August 2016

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Wenliang Du for all the guidance and support during my PhD study. It has been a pleasure and a privilege to work and learn from him. His patience, insights and constructive feedbacks have helped me formulate my dissertation topic and progress in my PhD research. Discussing new ideas is the most exciting and fruitful experience I had in my PhD study.

I thank Prof. Riyad S. Aboutaha, Prof. Kishan Mehrotra, Prof. Edmund S. Yu, Prof. Vir V. Phoha and Prof. Heng Yin for agreeing to be on my thesis committee. I am grateful to them for their time and efforts for helping me succeed in my PhD dissertation. Special thanks go to Prof. Heng Yin for the great learning experience I had while taking research classes under him.

Most of all, I would like to thank my beloved parents, my sisters and my brother for their unconditional love, encouragement and support.

I have been very fortunate to collaborate with my colleagues Xiao Zhang, Amit Ahlawat, Hao Hao, Kailiang Ying and Paul Ratazzi. I would also like to thank Ezgi Gozen, Scarlett Davidson, Vicky Singh, Haichao Zhang and others for their great friendship.

I was fortunate to have the opportunity to collaborate with some great researchers outside of Syracuse University such as Prof. Xiaofeng Wang from Indiana University at Bloomington, Michael Grace and Xiaoyong Zhou at Samsung Research America.

TABLE OF CONTENTS

Page

ABSTRACT i

LIST OF TABLES ix

LIST OF FIGURES x

1 Introduction 1

1.1 Android Customization Process 2

1.2 Android Customization Hazards 4

1.3 Thesis Statement and Contributions 5

1.4 Dissertation Roadmap 7

2 Android Customization 9

2.1 Fragmentation Factors & Parties 9

2.1.1 Custom Hardware Components 9

2.1.2 Device Vendor Specific User Interface 10

2.1.3 Device Vendors Specific Additions and Services 10

2.1.4 Operators Specific Features 11

2.1.5 Android Updates 12

2.2 Customization Effects 12

2.2.1 Compatibility and Portability Issues 13

2.2.2 Android Updates Problem 15

2.3 Systematic Categorization of Android Customization 16

2.3.1 Android Layered Architecture 16

2.3.2 Customization Aspects: Application Layer 18

2.3.3 Customization Aspects: Framework Layer 20

2.3.4 Customization Aspects: Kernel Layer. 22

2.4 Customization Security Risks Categorization 24

Page

2.4.1 Security Risks due to Additions . . . 24
2.4.2 Security Risks due to Modifications . . . 26
2.4.3 Security Risks due to Removal . . . 29
2.4.4 Summary of the Security Risks . . . 29
3 Related Works . . . 32
3.1 Security Hazards of Android Customization . . . 32
3.2 Demystifying Android Security . . . 37
3.3 Android Vulnerabilities. . . . 43
3.4 Android Malware. . . . 47
4 Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References . . . 51
4.1 Exploiting Hares . . . 56
4.1.1 Package, Action and Activity Hijacking . . . 58
4.1.2 Content-Provider Capture . . . 63
4.1.3 Permission Seizure . . . 69
4.2 Detection and Measurement . . . 71
4.2.1 Harehunter . . . 71
4.2.2 A Large-scale Measurement Study . . . 78
4.2.3 App-level Protection . . . 84
4.3 Discussion . . . 86
5 Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis . . . 89
5.1 Investigation & Methodology . . . 93
5.2 Feature Extraction . . . 94
5.2.1 Permissions . . . 97
5.2.2 GIDs . . . 98
5.2.3 Protected Broadcasts . . . 100
5.2.4 Component Visibility . . . 101
5.2.5 Component Protection . . . 103
5.3 Data Generation . . . 104

Page

5.4 Differential Analysis . . . 105
5.5 Results and Findings . . . 111
5.5.1 Overall Results . . . 112
5.5.2 Permissions Changes Pattern . . . 114
5.5.3 Permission-GID Mapping . . . 116
5.5.4 Protected Broadcasts Changes Pattern . . . 117
5.5.5 Component Security Changes Pattern . . . 117
5.5.6 Downgrades Through Version Analysis . . . 121
5.6 Attacks . . . 122
5.7 Limitations . . . 127
6 Conclusion and Future Work . . . 129
A DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android . . . 131
A.1 Approach Overview . . . 133
A.2 Feature Extraction and Refinement . . . 135
A.2.1 Extraction of Dangerous APIs . . . 136
A.2.2 Extraction of Package Level Information . . . 136
A.2.3 Extraction of APIs Parameters . . . 137
A.3 Insights in API-Level Malware Behavior . . . 138
A.3.1 Application-specific resources APIs . . . 139
A.3.2 Android framework resources APIs . . . 141
A.3.3 DVM related resources APIs . . . 142
A.3.4 System resources APIs . . . 142
A.3.5 Utilities APIs . . . 144
A.3.6 Parameters Features: . . . 145
A.4 Classification and Evaluation . . . 147
A.4.1 Data Set . . . 147
A.4.2 Classification Models . . . 147
A.4.3 Permission-Based Feature Set . . . 149

Page

A.4.4 API-Based Feature Set with Package Level and Parameter Information 150

A.4.5 Models Comparison 152

A.4.6 Processing Time 153

A.5 Discussion 155

A.6 Conclusion 156

LIST OF REFERENCES 157

VITA 167

LIST OF TABLES

Table		Page
4.1	Android Images Collected	78
4.2	Hares Prevalence in System Apps per Vendor	79
4.3	Possible Impact of 33 Randomly Picked Hares	82
4.4	Hare Flaws in Different Vendor A Models Running Android 4.4.2	82
4.5	Hares in Phone 3 Running Android 4.4.2 For Different Countries and Carriers	83
5.1	Security Checks	98
5.2	Collected Android Images	105
5.3	Security Configurations Map	106
5.4	Drivers accessible to System GID	125
5.5	Impact of Inconsistent Security Configurations	127
A.1	Categorization of Parameters to Frequently Used Malware APIs	138
A.2	Some Frequent API Parameters in Malware	146
A.3	Processing Overhead of the Classification Algorithms	154

LIST OF FIGURES

Figure		Page
2.1	Android Layered Architecture and Components	17
2.2	Customization Security hazards due to addition of new entities	24
2.3	Customization Security hazards due to modifying existing entities	26
4.1	Exploiting a Hare Authority to Hijack Email Account Settings Activity	66
4.2	Design of Harehunter.	73
4.3	Signature Based Guard Example	76
4.4	Feature Based Guard Example	76
4.5	Distribution of Hares across Different Hare Categories	80
4.6	Ratios of Vulnerable Apps Across Different OS Versions and Manufacturers	83
5.1	Investigation Flow	94
5.2	Android Security Model	96
5.3	Overall Inconsistencies Detected A1: Cross-Version, A2: Cross-Vendor, A3: Cross-Model, A4: Cross-Carrier, A5: Cross-Region	111
5.4	Protection Level Changes Patterns	115
5.5	Inconsistency Breakdown	116
5.6	Breaking Down Components: Visibility Mismatch	118
5.7	Breaking Down Components: Permission Protection Mismatch	119
5.8	Percentage of Security Features Downgrades	122
A.1	Our Approach	134
A.2	Top 20 APIs with the Highest Difference Between Malware and Benign Apps	139
A.3	Performance of Permission-based Models	150
A.4	Performance of API-based Models	151
A.5	Models Comparison	153
A.6	Distribution of DroidAPIMiner Processing time	154

1. INTRODUCTION

The smartphone market has expanded explosively in recent years as more consumers are attracted to the rich features, sensor enabled devices. Existing smartphones are not simply devices for making phone calls and receiving SMS messages, but powerful communication and entertainment platforms for web surfing, social networking, GPS navigation and online banking. Android is particularly eminent; never before any OS has been so popular and diverse as Android. So far, over one billion mobile devices are running the OS, occupying a smartphone market share of 82.8% as of 2015 according a report from IDC [1]. The popularity of Android is spurred by its open ecosystem nature that Google adopted to tear down some of the barriers caused by the closed nature of traditional operating systems. By being an open ecosystem, Android aims to encourage innovation and to bring more choice and empowerment to end users, which has been indeed embodied in the proliferation of feature rich devices as well as compelling mobile applications (apps). The open nature of Android ecosystem has naturally laid the foundation for a highly fragmented operating system. In fact, the official versions (Android Open Source Project or AOSP) have been aggressively customized into thousands of system images by everyone in the customization chain. Hardware manufacturers, device vendors and carriers are free to build upon the baseline to tailer it for different features and models in a bid to differentiate their products from their competitors. As device vendors are more adept at changing the Android framework and default system apps, and as hardware have grown more capable (i.e. high

resolution cameras, innovative sensors, etc), Android customization is becoming more prevalent overtime. In fact, according to OpenSignal [2], 2015 has seen an increased fragmentation of manufacturers, with over 1000 manufacturers not seen in the year 2012.

However, if not well thought out, this customization could bring in serious security vulnerabilities, causing severe damages. Indeed, previous studies [3–5] demonstrate several vulnerabilities caused by the unregulated Android customization process, ranging from over-privileged preloaded apps, information leakage, misconfigured Linux layer device drivers, etc. Given the serious damages and the wide consumer audience at stake of any security flaw caused by Android customization, it is important to study the aspects of this practice and investigate any potentially uncovered security consequences. The study will help draw a wider picture of Android customization with regards to its security effects and assess the overall situation in real world custom images.

1.1 Android Customization Process

Android is a layered operating system, where each layer has its own tasks and responsibilities. Different parties in the customization chain such as device vendors (e.g. Samsung, HTC, Xiaomi), carriers (e.g T-Mobile, AT&T), and hardware manufacturers (e.g. Qualcomm, mediaTek) might customize one or more layers to tailer the devices for different purposes, such as supporting special hardware and providing different interfaces and services. The application layer, lying at the top of Android architecture model and providing core apps such as Home, Contact, Phone and Browsers, is often customized by device vendors and carriers. Previous work [4] has demonstrated that this layer has always

been the focus of vendor customization. Device vendors and carriers might add new preloaded apps to support new services, modify default preloaded AOSP apps to provide more sophisticated functionalities and UI, or delete existing ones (e.g remove the Messaging app if the device is a tablet model).

The Framework and Libraries layers lie right below the Application layer and provide support for developers to access various privileged resources, services and functionalities. The previous study [6] makes the case based on the analysis of a custom phone (HTC EVO 4G) that Android Framework and Library are indeed extensively customized by the vendor. The study reveals that certain framework binaries have been modified: core.dex which contains the core Java language public APIs and other popular libraries such as Apache Harmony Library; framework.dex which constitutes the core library for the Android framework; services.dex that hosts a number of services such as PackageManager Service, ActivityManager Service, WindowManager Service, etc; as well as android.policy.dex which enforces device security policies such as mandatory screen lock.

At the bottom of the layers is the Linux kernel, providing a level of abstraction between the device hardware and containing all essential hardware drivers like display, camera, etc. Hardware manufacturers almost always need to modify a few device drivers and related system settings to support any newly added hardware. The previous work [5] reveals that most of the customization at this layer is actually related to device driver customization.

1.2 Android Customization Hazards

Android device customization implies an opportunity for increased usability and personalization as it empowers users to choose the Android device that perfectly meets their specific needs. However, it also implies an increased security risk, given the Android fragmentation process is highly unregulated. To put this under control, Google has launched the Android Compatibility Program [7] to guide the customization process. Nonetheless, this effort fails to address several security concerns arising from the aggressive customization. The existing literature on Android customization hazards reports various vulnerabilities. At the Linux kernel layer, the study [5] investigates the security configurations of Android's Linux device drivers, and finds that many of these devices have not been properly protected, causing their exposures to the parties that should not access them: e.g. a non-privileged app (without the required camera permission) can directly command an exposed camera driver to take pictures.

Another study [4] reveals that the Application layer is also riddled with flaws introduced during the customization process. Based on the analysis of preloaded system apps from 10 stock factory images manufactured by different vendors, the study reports the presence of known vulnerabilities such as over-privileged apps, exposed interfaces leading to permission re-delegation attacks in system applications introduced by the corresponding vendors. Similarly, another work [3] has anecdotally shown that Android vendors' preloaded apps had security flaws shipped on several custom devices.

These studies demonstrate few serious security vulnerabilities resulting from Android customization. However, little has been done to systematically study this process, its

involved parties and carried-on changes to asses all possible security consequences. This dissertation attempts to perform a systematic analysis of the decentralized and under-regulated Android device customization process and consequently uncover possible security risks, never investigated before. For this purpose, I first study several aspects of Android customization, including its causes, stakeholders, carried-on changes and finally speculate potential security effects if the customization is not well thought out. I then conduct further investigation to prove my speculations with real world attacks. My investigation lead to the discovery of a novel security vulnerability, Hanging Attribute References (HARE), which I prove to be security critical and extensive (details are in Chapter 4). I further identify a more general security risk that spans over all Android layers, resulting from non-carefully modifying several security configuration features during the customization process (details are in Chapter 5).

1.3 Thesis Statement and Contributions

The thesis statement of this dissertation is as follows: **To proactively uncover potential security risks of the decentralized and unregulated Android customization process, this dissertation performs a systematic investigation of inconsistencies created as a result of this process and assesses various security implications**. In support of this statement, this dissertation describes the following contributions:

- Hanging Attributes References Threat: This dissertation investigates a critical aspect of Android device customization overlooked in previous studies and whose

implications are not clear. During customization, preloaded Android apps are modified, added, or deleted to tailor custom devices for different purposes. For example, a vendor may customize a smartphone OS without a 3G capability by removing some components including the messaging and telephony provider apps. However, other components such as VoIP apps might still reference the removed apps. Thus, customization made to these components, if not well thought out, can break the intrinsic relations that exist between Android's components, resulting in references to non-existing attributes. We call them Hanging Attribute References, or simply Hares. A malware can fill the gap to acquire critical system capabilities, by simply disguising as the owner of the removed component. Through performing a large scale study, this dissertation shows that popular Android devices are riddled with such flaws, which often have serious security implications leading to real-world attacks (stealing user voice notes, replacing Google Email's Account Settings Activity, collecting user's contacts without proper permissions, etc). We further designed and implemented Harehunter, a new tool for automatic detection of Hares that compares attributes defined with those used, and analyzes the references to undefined attributes to determine whether they have been protected.

- Harvesting Inconsistent Security Configurations in custom Android ROMs: My thread of research moved from trying to reveal specific vulnerabilities that might be introduced as a result of Android customization to generalizing the problem to a broader scope. More specifically, in this dissertation, I propose to systematically detect security configuration changes introduced by different parties in the Android

customization chain. My key intuition is that through comparing a custom device to similar devices from other vendors, carriers, regions, or OS versions, we can identify security configuration changes created unintentionally during the customization. For this purpose, I first locate relevant security features that might be altered during this process through investigating the Android access control model at different layers. Then, I extract them from my collected Android custom ROMs (around 600 ROMs). To detect inconsistent security features, I perform a differential analysis among candidate image sets sharing similar features. My differential analysis reveals that indeed the customization results in inconsistent security configurations, and more dangerously, sometimes weaker security features. For example, my investigation shows that vendors sometimes downgrade the privilege of Linux group ids, making normal apps able to access very privileged resources on the system with a normal permission. Besides, vendors sometimes downgrade the protection level of built-in permissions leading to conflicting definitions throughout different images. This dissertation also proves that the security configuration inconsistencies revealed in my analysis can lead to actual vulnerabilities, including factory resetting the phone, sending emergency broadcasts, and reading user emails, all without user confirmation or a privileged permission.

1.4 Dissertation Roadmap

The rest of this dissertation is organized as follows: Chapter 2 provides an overview of Android customization process, its factors, responsible parties and possible effects.

Chapter 3 reviews related literature on Android customization and on demystifying Android security in general. Chapter 4 presents my discovered Hanging Attribute References (Hare) vulnerability, while Chapter 5 presents my work on harvesting inconsistent security configurations. In the end, Chapter 6 summarizes this dissertation. In a different direction, Appendix A presents my research work, DroidAPIMiner, on Detecting Android malware.

2. ANDROID CUSTOMIZATION

The open nature of Android ecosystem, promoting manufacturer differentiation and device innovation, has naturally laid the foundation for a highly fragmented ecosystem. Hardware and chipset manufacturers customize Android devices to provide different performance levels, hardware components and screen sizes. Vendors further customize the Android framework to support more sophisticated features and offer a unique user-experience. Further complicating the process, the AOSP baseline has been heavily modified into different Android versions. According to a survey by app maker and network data collector OpenSignal [2], the year 2015 has seen more Android devices and more differentiation between devices in the market. Out of the 682,000 devices surveyed, more than 24,000 distinct devices and 1294 distinct brands have been identified, where the largest portion (37.8%) were manufactured by Samsung.

2.1 Fragmentation Factors & Parties

2.1.1 Custom Hardware Components

At any moment, Android devices might be running different hardware components (e.g., processors, graphic cards, and screen sizes). Hardware and chipset manufacturers customize Android devices to provide better performance levels and more sophisticated hardware. For example, Samsung Galaxy S7 Edge's 12-megapixel dual-pixel sensor enables

super fast autofocus, brighter and sharper photos even in low-light conditions, while LG G5's second wide-angle camera enables stunning landscape photography. With each new model release, device vendors often add more and more sensors to differentiate their devices from other vendors. To illustrate, in newer models (Galaxy Note 4, Galaxy S6 Edge, etc), Samsung integrated a high recognition fingerprint scanner, and a new heart rate monitor sensor. Vendors adapt the basic Android software stack to the new vendor-specific hardware platforms through integrating several additions to the Android OS.

2.1.2 Device Vendor Specific User Interface

Android's User Interface is everything that the user is seeing and interacting with, and clearly one of the key differentiating factors for an Android device. Thus, device vendors strive to provide a richer and unique user experience enabling more intuitive and esthetically enhanced interfaces. In fact, several vendors design their own differentiated UI. To name a few, Samsung's UI is branded as TouchWiz, HTC's UI is branded as HTC-Sense, and LG's UI is branded as Optimus. Each custom UI uses its own color palates and UI elements (e.g., buttons and scrolling lists) to present aesthetically dissimilar arrangements. Device vendors integrate these custom UIs into their own devices through adding and altering the existing Android Stock UI elements.

2.1.3 Device Vendors Specific Additions and Services

Device vendors might customize Android to provide business solutions or personalized services. A prominent example is Samsung's KNOX platform [8] which leverages hardware

security capabilities to offer multiple levels of protection for the Android OS and installed apps. The KNOX platform offers also an easy integration with popular Mobile Device Management, Single Sign-On, and VPN solutions. To provide the KNOX security platform, Samsung has integrated several solutions in the Android OS, such as Trusted Boot, TrustZone based Integrity Measurement Architecture, and SEAndroid.

2.1.4 Operators Specific Features

The Android customization process continues when the device is tailored for a specific network operator (i.e., mobile carrier), such as Verizon, AT&T, Sprint, Orange, etc. The device vendors (i.e., Samsung, HTC, etc.) are obliged to comply with the operator's published network specifications, which are usually related to some network configurations nuances. Based on carrier network requirements, device vendors modify the telephony service to allow integrating a variety of LTE, GSM/CDMA bands.

Device vendors have to also conduct a series of changes, often spanning through several Android layers, to support specific operator restrictions. Network operators might intentionally require disabling some built-in Android features, following their business model. In fact, major operators disable Android's native tethering support, hotspot support, and other features related to network sharing. Other operators, such as AT&T, have disabled in few models the "Unknown sources" checkbox that allows installing apps from unofficial markets such as the Amazon App Store and others. Operators might also disable access to competing services such as blocking digital wallet solutions, forcing users to use their supplied services. More customization might be carried out by device vendors

to provide the support for unique operator services. For example, most operators provide their own custom voice mail services, such as "Visual Voice Mail" service provided by Verizon which lets users view voice messages received and listening to them in any order. Integrating these new services into the existing main stream Android usually requires device vendors to modify one or more Android layers.

2.1.5 Android Updates

Further complicating the Android customization process is the fast pace with which the AOSP updates its OS versions. Since September 2008, 23 (from 1.0 to 6.0.1) official versions have been released. Most of them have been heavily customized, which results in tens of thousands of customized Android branches coexisting on billions of mobile phones around the world (over 10,000 for Samsung alone), and many of them, at various version levels, co-exist in today's market. Making the situation more complex, to upgrade to a new version, device vendors have to migrate all carried out customization changes at various levels before a public release

2.2 Customization Effects

The above customization factors have led to a highly fragmented ecosystem, which not only make development and testing of new apps across different device models and specifications a challenge, but also inevitably open the door for a plethora of security risks. When hardware manufacturers, device vendors and carriers perform their customization

without fully understanding the security implications, security vulnerabilities will arise, at various Android layers.

2.2.1 Compatibility and Portability Issues

Because of the differences between Android releases, hardware specifications, and device variations (e.g, different screen sizes, etc.), the effort required to build applications that work seamlessly on all devices can be exhausting; which might intuitively introduce compatibility issues.

From one hand, each new Android API level introduces or removes features and bugs. Even if each release aims to integrate the new changes without breaking existing pre-installed apps in older versions, it is often not feasible to achieve perfect compatibility. Due to insufficient cross-platform and product-lines testing of the new features and changes, Android apps might not behave consistently across different versions. In an effort to provide empirical evidence about the features that contribute most to Android fragmentation, Han et al [9] investigate bug reports submitted by Android users of two major vendors (HTC and Motorola). Their study reveals that bugs related to upgrades occurred quite frequently when users moved from Android version 2.1 to 2.2, indicating that the latter introduced compatibly and portability issues.

On the other hand, hardware specific variations and features might introduce unique compatibility issues within different product-lines of the same vendor. In fact, the same study [9] reports that different product-lines produced different bug report topics indicating internal fragmentation issues within the same vendor. The portability issues are

attributed to varying hardware components in the studied vendors' models, such as screen sizes, buttons behavior, gps, wifi, and Bluetooth support .

To ensure a consistent and correct behavior of Android apps across various API levels and hardware configurations, Google introduced Firebase Test Lab for Android [10], a cloud-based infrastructure for comprehensive testing of Android apps before release. Using Firebase Test Lab, developers can access various physical Android devices installed on Google data centers and test their apps on different brands and models, across multiple Android APIs, device configurations, and screen orientations. Unlike testing apps on Android emulators, Firebase Test Lab tests the apps on physical devices to help detect issues that might not occur in an emulated environment.

Besides, to address the fragmentation and to ensure compatibility across Android devices, Google launched earlier its Compatibility Program [7]. The program aims to benefit the entire Android community, including users, developers, and device manufacturers. More specifically, first, the program aims to provide a precise definition of what developers can expect from a given device in terms of APIs and hardware capabilities. Second, through providing an appropriate application filtering channel (such as Google Play), Android users will be presented only with the apps that are compatible with their device's hardware and software capabilities. Finally, the program focuses on allowing devices manufacturers to create devices that are unique but nonetheless compatible with Android aspects relevant to running third-party apps.

2.2.2 Android Updates Problem

Each month, Google advises security updates to be pushed on devices to fix known and detected security vulnerabilities and bugs. However, device manufacturers and carriers might be busy testing and optimizing the security updates for their own customized versions of Android. This produces huge delays in following Google's patch schedule and in pushing the advised updates to existing device models. Even worse, device vendors might opt to stop pushing further security updates to older models due to a lack of monetary incentives, a fact that can make several unpatched custom android devices under major security threats. For example, Samsung's Note 2.0 have not received any security updates since 4.4.2.

The security implications of Android's updates cycle have been studied by the research work [11]. Using a corpus of 20400 custom Android devices, the authors demonstrate that there is a significant variability in delivering security updates across different device manufacturers and network operators. The study further reveals that 87.7% of the collected Android devices have major security vulnerabilities and are exposed to at least 1 major threat, including udev exploid [12], Gingerbreak [13], Apk duplicate file names [14], Apk unchecked names [15], etc.

To face this alarming update challenge, Google has committed that its Nexus devices will receive regular over-the-air security updates every month, as well as platform updates [16]. Similarly, Samsung has committed to implement a new Android security update process that fast tracks the security patches over-the-air when security vulnerabilities are uncovered in a more timely manner, about once per month [17].

Delayed Android updates is one of the many possible security issues resulting from Android customization. In an effort to draw a complete picture of possible security consequences, I propose to perform in the next section a systematic categorization of Android customization aspects. The categorization will help us pinpoint areas of concerns that might bring in security problems.

2.3 Systematic Categorization of Android Customization

In this section, I aim to systematically categorize the customization aspects discussed in section 2.1 with regards to the Android layered architecture. For each layer, I list all possible modifications performed by the customization stakeholders: device vendors, carriers, and hardware manufacturers. Then, to uncover possible security consequences, I dive into more details about the customization aspects within each layer.

2.3.1 Android Layered Architecture

As Figure 2.1 illustrates, Android is a layered operating system, where each layer has its own tasks and responsibilities.

On the top of all layers is the application layer. It comprises preloaded apps provided by AOSP (such as built-in web browser, messaging, email apps), by device vendors (such as Samsung's custom Settings, Samsung's Gallery app), and sometimes by carriers (such as Verizon's Visual Voice app). This layer also contains third-party apps installed by the user after purchasing the device.

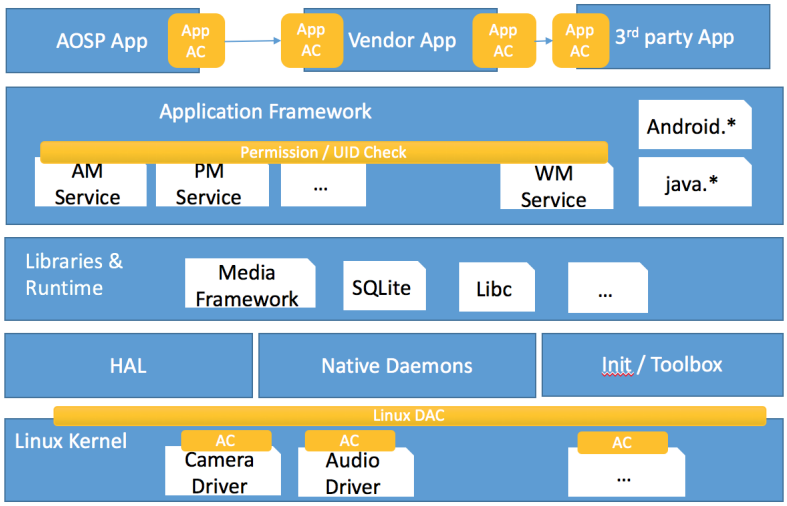


Fig. 2.1.: Android Layered Architecture and Components

To allow app developers to access various resources and functionalities, Android Framework layer provides many high-level services such as PackageManager, ActivityManager, NotificationManager and many others. These services mediate access to system resources and enforce proper access control based on several criteria such as the app's user id, its acquired Android permissions, signature, etc. Right below the framework layer lies the Libraries layer, which is a set of Android specific libraries and other core 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 similar criteria (e.i, the caller's user id, permissions, etc).

At the bottom of the Android layers lies the Linux kernel which provides a level of abstraction between the device hardware and the upper layers. It contains all lower level core system functionalities such as memory management, process and power management and provides essential hardware drivers like camera, device display, Wifi, etc. The Linux

kernel layer mediates access to hardware drivers and raw resources based on the standard Discretionary Access Control (DAC).

2.3.2 Customization Aspects: Application Layer

Customization stakeholders might customize the Android application layer in several aspects; they might add new preloaded apps, modify existing ones and / or remove unnecessary ones.

Adding new preloaded apps. In order to differentiate their products from other devices and provide more features, device vendors and carriers might ship new preloaded apps within their devices. In fact, Wu et al [4] performed a provenance analysis aiming to classify each preloaded app into three categories: apps originating from the AOSP, apps customized or written by vendors, and other apps bundled into the stock image. A breakdown of their results shows that on average, at most 18.22% of the apps originate from AOSP, implying that 81.78% of the preloaded apps are actually added by vendors and other parties (e.g., carriers).

Modifying existing AOSP and older editions of vendor specific apps. Existing AOSP apps may well be customized by device vendors to provide additional or richer functionalities and different UI layouts, as every vendor has incentives to add more functionality, especially if their competitors are doing the same. During the customization process, vendors might add new components (i.e., activities, services, broadcast receivers, content providers, and custom permission definitions) to existing preloaded apps to provide

new functionalities and services. For example, the Gmail app is actually a customized version of the original AOSP email app "com.android.email", where several components have been added. Similarly, during the customization, vendors might decide to omit certain components from existing AOSP apps that are deemed not necessary or that have been replaced with more sophisticated ones. To tailor the app for different needs, vendors might also change the implementation of one or more components.

Another common practice, observed during the customization process is renaming existing AOSP app names, package names, and component names with new names reflecting the vendors' identity. For example, Sony's Conversations app with package name "com.sonyericsson.conversations" is actually a customized version of the AOSP Mms app with name "com.android.mms". The new naming convention could also be attributed to the namespace requirements defined by Google's Compatibility Document [18], enumerating criteria that must be met in order for devices to be compatible with Android 6.0. In fact, one of the requirement states that if a device vendor is to add a new permission, Google recommends that its ID string should not be in the "android.*" namespace. To meet this recommendation, device vendors need to change their added custom permissions whose ID string starts with "android." to new names. The document also recommends that device implementers may introduce intent patterns using namespaces that are clearly and obviously associated with their own organizations. This has pushed device vendors to change their custom intent patterns to reflect their organization namespaces.

Another possible customization practice that might be carried out by vendors is changing access configurations of existing components in AOSP apps or their already

published preloaded apps. To restrict access to provided functionalities, vendors might decide to make certain components private to the app itself, or only accessible to other apps signed by them. In contrast, they might decide to expose or to require less strict permissions to access downgraded components in their modified apps. If not carefully carried out, this practice can open serious security holes.

Additionally, vendors might add new resources to existing AOSP and their already published preloaded apps. For example, they might add more shared preferences and files depending on the required functionalities. They might also change access control rights of existing resources for specific purposes.

Removing existing preloaded apps. Vendors may customize Android devices through removing certain apps that are not required for the device's functionality or that may have been replaced with a vendor specific one. For example, a manufacturer may customize a smartphone OS for a tablet without 3G capability by removing some apps related to telephony and messaging services, including the dialer app, phone app, telephony providers and Sms/Mms app.

2.3.3 Customization Aspects: Framework Layer

To incorporate new services and provide support for added functionalities, vendors often customize the Android framework layer and libraries. They might add new framework services and libraries, alter existing ones though customizing their implementation and adding new APIs to provide developers access to new resources and functionalities.

Adding new system services. Android system services play a key role in exposing low-level functionalities (e.g, hardware resources) to high-level applications, and in providing applications with the information and capabilities necessary to achieve desired functionalities. There are around 70 system services in the Marshmalllow release of Android, including PackageManagerService, ActivityManagerService and WindowManagerService, etc. To allow exposing new functionalities, vendors integrate new system services within the Android framework. In fact, according to our study [19] on 606 custom Android images manufactured by 11 different vendors, on average, vendors added 130 new services, where the largest number of added system services is reported in Samsung, Amazon and LG images. Supporting the same observation, the study [6] which proposes to compare an HTC device (HTC Evo 4G) to the official Android AOSP and to locate the manufacturer's modifications, reports that Android system services binary (services.dex) is one of the most heavily customized binaries in Android. In the analyzed device, several system services have been added for HTC pen support, USB-based networking, and for integrating a proprietary logging facility.

Modifying existing system services. To tailor existing system services for new functionalities, manufacturers might modify existing system services. In fact, based on the same study [6], 95 new classes and 1139 methods have been added while 124 classes and 384 methods have been modified in services.dex extracted from the analyzed HTC Evo 4G device.

Modifying other framework binaries. Similarly, vendors might modify other framework binaries to provide support for added functionalities. The same study [6] reveals that android.policy.dex, containing enforcement of device security policies such as mandatory screen lock, has been modified to change the user interface of the lock screen and to support a new introduced feature (e.g., touch-based stylus). core.dex and ext.dex also manifest few Java libraries additions, while framework.dex shows extensive customization; more than 1500 classes have been added, and more than 1100 classes have been modified.

Changing configuration files. Android framework contains several configuration files that might be customized by vendors for specific reasons. Framework resources apks (located under /system/framework) contain the definition of most built-in permissions and protected broadcasts. Vendors may customize these configuration files to add new permissions, modify existing ones, add new protected broadcasts or remove existing ones. Other framework configuration files are located under /system/etc/ and contain critical system wide configurations. Vendors might modify platform.xml to customize gid (Linux group Id) to permission mappings or change other configurations under the same directory.

2.3.4 Customization Aspects: Kernel Layer.

Hardware manufacturers, such as Qualcomm and MediaTek, often customize the Android Kernel layer to fit their hardware components. They often add new device drivers or modify configurations of existing ones.

Adding new device drivers. Android inherits the way Linux manages its device drivers in which block and stream devices are mostly placed under /dev. By default, Android introduces a set of device drivers, such as GPS, audio, camera, etc. In addition to that, vendors often introduce a plethora of new devices drivers for custom CPU, graphic devices, etc. In fact, one of the primary reasons for vendors to customize Android is to make it work with added hardware. Thus, vendors need to fit the corresponding new device drivers to the AOSP baseline. As device drivers communicate with Android users through framework services such as the Sensor Service and the LocationManagerService, any customization made to the device drivers should be properly propagated to the corresponding framework services. Thus, device vendors often need to customize these services to support the custom hardware components.

Removing device drivers. Vendors might decide to remove certain device drivers if the corresponding hardware have been removed from a specific model. For example, a vendor might decide to remove NFC driver if the device does not include an NFC card.

Modifying configurations of existing device drivers. Hardware manufacturers often need to modify the configuration of existing device drivers (e.g., gps driver, audio driver, camera driver, etc.) and related configurations and settings to provide full support for the custom hardware. In fact, Zhou et al [5] conducted a large scale measurement study to understand the scope and magnitude of vendor device customization in 2423 factory images; the study led to the discovery of 1290 images containing at least one device file

whose protection level is set differently from the reference AOSP images (publicly readable and writable, below that of the reference device file).

2.4 Customization Security Risks Categorization

Based on my analysis of the customization aspects carried out at various Android layers, I can categorize customization security effects based on the following three categories: risks due to additions, risks due to modifications and risks due to removals.

2.4.1 Security Risks due to Additions

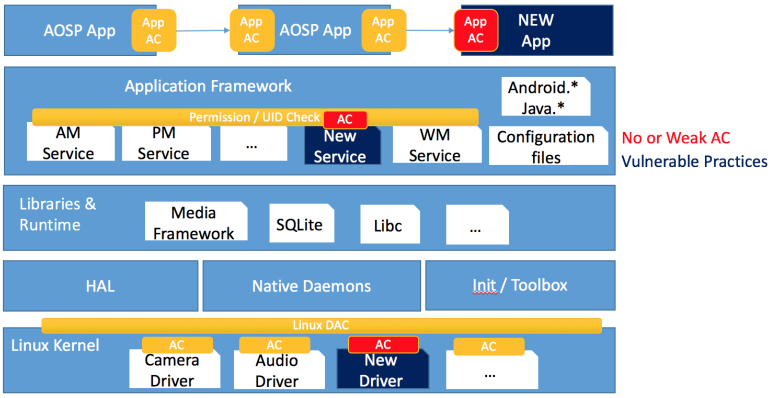


Fig. 2.2.: Customization Security hazards due to addition of new entities

Figure 2.2 summarizes possible security effects of introducing a new app, component, or framework services, libraries, and kernel drivers.

Buggy and Vulnerable System apps. If not carefully designed and implemented, newly introduced preloaded apps or components might contain known dangerous practices and even security vulnerabilities. In fact, the prominent work [4] demonstrates that the

majority of studied vendor preloaded apps exhibited permission over-privilege; that is, they request more permissions than what is actually needed by the apps, indicating a poor understanding of the Android security model. The same work shows that vendor apps contain know Android vulnerabilities. First, a large percentage of vendor introduced apps included permission re-delegation attacks [20], which are a form of the classic confused deputy attack [21] allowing an authorized app to gain access to an Android permission without actually requesting it. This is particularly more dangerous in the context of preloaded apps as it might lead to gaining system or signature level permissions, not available to third-party apps. Second, the study detected that a huge percentage of vendor apps contained passive content leaks and content pollution [22]. A content leak happens when a content provider is world-readable (i.e., one that takes no sensitive permission to protect its read access), or if it is accessible from another exposed component (e.g, activity, receiver, or service). A content pollution on the other hand, exists when a content provider is world-writable (e.g., not using a sensitive permission to protect its write access) allowing an unauthorized app to manipulate certain in-app data.

Vulnerable framework services and libraries. System services provide and intercept access to core Android functionalities, and are responsible of enforcing proper access control based on the caller's identity (e.g., UID, PID, package signature, etc.), caller's permissions, and other criteria. Omitting an access control check within vendor added system services' APIs might expose corresponding functionalities to non-authorized apps; thus leading to serious security holes. Similarly, employing weaker access control checks might also put the corresponding operations at risk, as it can be easily circumvented.

Weak configurations of new device drivers. Vendors are responsible of specifying the filesystem permission bits for their added device drivers, some of which are security-critical. These devices, if not properly protected, could allow an unauthorized app to access sensitive user data (e.g., GPS location, selected screen coordinates, etc.) or system capabilities that usually require dangerous / system Android permissions (e.g., redirecting the driver to take a picture or take a screenshot, etc). Thus, vendors might put the device at risk if they do not enforce proper access control permissions on newly added device drivers.

2.4.2 Security Risks due to Modifications

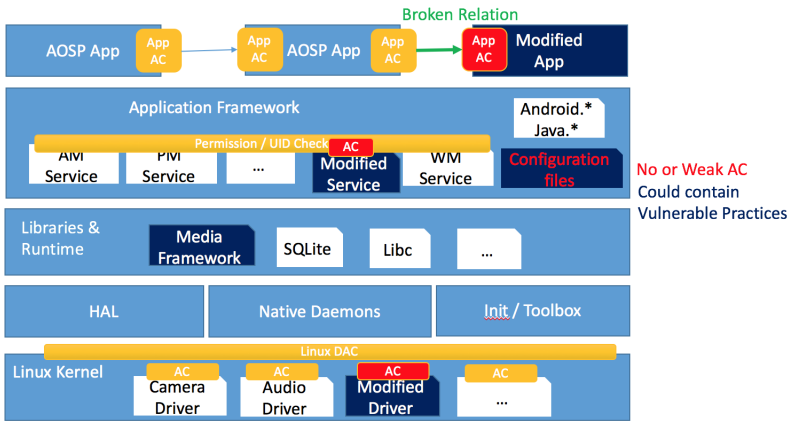


Fig. 2.3.: Customization Security hazards due to modifying existing entities

Figure 2.3 summarizes possible security effects of modifying existing preloaded apps' configurations or implementation, modifying framework system services and libraries and modifying system-wide settings, including device drivers.

Vulnerable System apps. During the customization of preloaded Android apps, vendors might weaken the access control on existing components through omitting security checks implemented programmatically (e.g., binder.getCallingUid API, context.checkCallingPermission API, etc.) or through downgrading their protection within the manifest files. For example, a developer might decide to expose a component that has been protected in older versions or to remove any required caller permissions. If the underlying component provides privileged capabilities, downgrading its protection would naturally lead to known Android vulnerabilities such as permission re-delegation attacks, and content leaks and pollution attacks.

Weaker System wide configurations. During the customization process, device manufacturers might decide to alter certain security-critical configurations used to protect privileged resources and capabilities on the system. If not carefully carried out, the new security configurations might be weaker than the original ones, thus breaking some of the assumptions made by other components. For example, through altering configurations within framework-res.apk, a vendor might unintentionally downgrade the protection level of system-wide Android permissions and remove protected broadcasts declarations. Similarly, through introducing wrong gid to permission mappings, vendors might downgrade the permission level required to obtain critical gids. If developers and other customization parties are not aware of these weaker configuration changes, they might continue to use them to protect highly privileged resources.

Weaker configurations of existing device drivers. Similar to the discussion earlier, device manufacturers might customize the security protection level of existing device drivers, which might be way weaker than their counterpart on the AOSP reference model. In fact, though analyzing device driver configurations of 2423 images and comparing them to their counterpart on AOSP, the study [5] identified 1290 likely vulnerable images, including at least one device driver whose write/read access are weaker than the same file on the reference AOSP. The authors demonstrated the consequences of exposing one critical device node to the public on Samsung Galaxy SII through designing an app that can command the camera driver to take pictures without having the required permissions.

Breaking relationships between Android components. Different Android components (apps or their internal activities, services, content providers, receivers, etc.) are connected together by Inter-Component Communication (ICC). Intents are the primary vehicle for ICC, and describe operations to be performed by the recipient. For example, to render the user's current GPS coordinates on a map, a developer can build an intent containing the target location and send it to any component that handles rendering maps. Developers can invoke intent recipients in two ways, explicitly where the target component's package name and class name are specified, or implicitly where the target's intent filters are set (e.g., action, category, data fields, etc.). Through modifying components' identifiers and intent filters' attributes during the customization, vendors are risking to break the ICC that exists intrinsically between different preloaded apps, installed third-party apps, and even framework services referring to other components on the device. If developers are not aware of the new identifiers' names, they might refer to non-existing

components through using previous identifiers (previous package names, class names, content providers authorities, older action filters, etc.). For example, to save voice notes to the default notes' database, a developer might use a previous identifier of the content provider authority, leading to a possible crash of the app or even worse leaking the voice notes to non-authorized parties. In fact, If the ICC reference points (i.e., invocation of the content provider's authority) are not well guarded (e.g., through a signature check), a malicious app might claim the ownership of the referenced entity and thus, acquire some privileges associated with it.

2.4.3 Security Risks due to Removal

Just like modifying existing Android components, removing a certain app or component from existing apps can lead to breaking the intrinsic relationship that exists between them. When an attribute (e.g. a package name, authority, action, etc.) is used on a device but the party defining it has been removed, a malicious app can fill the gap to acquire critical system capabilities, by simply disguising as the the owner of the attribute.

2.4.4 Summary of the Security Risks

My systematic categorization is not only able to cover all security risks already discovered on Android customization (as of my knowledge), but to also lay the ground for another investigation space aiming at studying other serious customization risks, including investigating aspects of breaking the relationships between Android components,

introducing vulnerable framework services, weakening the existing framework services' access control checks, and more generally, introducing weaker system-wide security features.

In this dissertation, I make the first step towards understanding and assessing the security effects of breaking the intrinsic relationship that exists between components as a result of Android customization. My investigation reveals a new vulnerability, never investigated before, that can take place once components are removed or renamed while other components are still referring to them. The reference to the non-existing components becomes hanging and could be exploited by malicious parties through claiming the non-existing attribute to acquire privileged capabilities and mount different attacks. I discuss more details about this customization-due vulnerability in Chapter 4 and prove (with other collaborators) that it is actually prevalent in major vendors and also security critical.

In the second part of this dissertation, I generalize my finding on Hanging Attribute References (Hare) vulnerability as well as my systematic analysis of other proved and potentially possible customization security risks. Existing Android literature demonstrates 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 re-delegation or permission leakage attacks. My work on Hanging Attribute References and the research work ADDICTED [5] show further serious risks allowing an attacker to mount severe phishing attacks and to access resources without the corresponding permission. All the problems reported so far on Android customization are mainly caused by vendors' altering of critical configurations. I have shown in my systematic categorization how vendors might carelessly change important configurations; they might change the

protection of components within preloaded apps leading to exposed interfaces and potentially to component hijacking attacks, alter attributes of existing identifiers (e.g., package names, authorities, etc.) and even remove them leading to the hanging attribute reference vulnerability. I have also pointed out that it could be possible that vendors might change other system-wide configurations such as gid to permission mappings, etc

Albeit the demonstrated serious consequences of altering security features configurations, no work has systematically investigated all security configuration changes caused by vendor customization. In this dissertation, I propose to systematically detect security configuration changes introduced by parties in the Android customization chain through a large scale differential analysis. My work identified a large number of inconsistent configurations indicating that vendor customization is highly under-regulated and requires more security scrutiny. I discuss more details about inconsistent security features harvesting in Chapter 5.

3. RELATED WORKS

The popularity of Android operating system has attracted a lot of interests in the recent years. The main research directions fall into three categories: understanding the security landscape of the Android ecosystem; proposing various solutions to enhance its security architectural model, and uncovering emerging threats and vulnerabilities at various Android layers (app, framework, and Linux kernel). This dissertation falls in the last category as it mainly identifies security threats caused by Android customization, systematically.

In this section, I first review prior research related to Android customization. Then, I review other studies with regards to these three categories: demystifying Android security, Android security enhancement proposals, and finally, uncovering emerging threats.

3.1 Security Hazards of Android Customization

The security risks introduced by the fragmented Android ecosystem has been studied before. The previous work [3], which systematically studies 8 popular Android smartphones from different manufacturers reveals that these phone images do not properly enforce the permission-based security model. Several privileged or dangerous permissions protecting sensitive user data within preloaded apps are exposed to non-privileged apps. The authors developed Woodpecker, a tool that allows detecting capability leaks in

Android, and employed it to identify vulnerable cases in few customized Android devices. For example, the tool revealed that a preloaded app in Samsung's Epic 4G contains an explicit leak of the permission MASTER_CLEAR, which once exploited, allows an attacker to wipe the user data on the phone without his confirmation. Similarly, the tool discovered that the studied HTC's messaging app contains another serious capability leak, exposing the SEND_SMS permission to third party apps, and consequently allowing them to send sms messages to premium numbers. Even though this research work has just anecdotally shown that Android devices have security flaws shipped in their pre-loaded apps, it is one of the earliest to provide insights on the dangers of vendor customization.

Prior research [4] aims to assess the impact of Android customization on the overall Android security at the application level and to determine the source of security risks troubling the security aspects of Android preloaded apps. For this purpose, the authors developed a three-stage process; the first stage aims to classify each preloaded app into apps originating from AOSP, vendors, or other origins (e.g., carriers, etc.). The second stage analyzes the permission usage trend to detect over-privileged cases; that is, apps requesting more permissions than what they actually need for proper functioning. The third phase conducts a vulnerability analysis to detect permission re-delegation attacks [20] and content leaks and pollution attacks [22]; where permission re-delegation attacks allow unprivileged apps to act as though they have sensitive permissions via exploiting open interfaces of preloaded apps, and where content leaks allow such apps to gain unauthorized access to private data. The evaluation results of this 3 stage process show that 81.78% of pre-loaded apps on stock Android devices are originating from vendor customization and are responsible for the majority of detected security problems (85.78%) (i.e., permission

over-privilege, permission re-delegation, etc). Unlike these prior studies, My HareHunter work (chapter 4) discusses new type of vulnerabilities never reported before, hanging attribute references vulnerability, which is also specific to the customization process, affecting mostly the Android application layer. My research demonstrates the serious consequences of the new flaws and identifies the fundamental cause behind them, which has never been done before. Moreover, My DroidDiff (Chapter 5) is fundamentally different from the above two works that aim to find specific known vulnerabilities (e.g., capability leaks, permission re-delegation, etc.) on different customized images through conducting a reachability analysis from an open entry point to privileged sinks within decompiled system apps. Instead, I leverage a differential analysis to point out inconsistencies in Android components' protection, and consequently detect unintentionally exposed ones. Moreover, my analysis gives further insights about possible reasons behind the exposure.

Another closely related study is the prominent work [5] analyzing security configurations of Android's Linux device drivers in an effort to discover fragmentation perils at the Kernel layer. The study performs a systematic study on the security hazards of Android device customization through automatically identifying the Linux files related to the operations on a given device driver (e.g., audio, camera, wifi drivers, etc.) and then comparing their protection levels (Linux file permission bits for each individual file) on a vendor's version with those on the corresponding AOSP version. Any detected weaker protection on a vendor device driver implies a potential security hazard. The presence of the discrepancy of Linux file permissions across two similar OSes, together with its relation with a dangerous Android permission (e.g., CAMERA permission) is quite alarming. Surprisingly, the research finds that many vendor specific device drivers have not been

properly protected, causing their exposures to the parties that should not access them: e.g., an app can directly command an exposed camera driver to take pictures, even when it does not have the camera permission. The research further conducts a measurement study reporting the pervasiveness of the problem across over 2,000 factory images; around 1290 (53.24%) analyzed images contain at least one likely CF, publicly exposing different device nodes such as GPU, Unified Memory Provider, 2D Graphics, camera and RFID device drivers. By comparison, my study on Hanging Attribute References (Chapter 4) happens on pre-installed apps, which requires more complicated code analysis than a simple check of Linux drivers' security configurations, as does this work. Pre-installed apps are known to be the main target of a customization [4] and their customization - specific flaws have never been investigated thoroughly before, to the best of my knowledge. On a similar track, my DroidDiff study (Chapter 5) have been inspired by this work, which is only limited to investigating Linux device drivers inconsistencies, to generalize the search space for security inconsistencies to all Android layers. From a top-down approach, this related work is actually a specific case of system-wide inconsistent security configurations. My finding on inconsistent GID to permission mappings (section 5.5.3) demonstrates another way that can expose critical device drivers. More specifically, through introducing a low protection-level permission mapping to privileged GIDs (e.g., a normal protection level permission mapping to CAMERA gid), a vendor will allow third party apps to gain access to security critical drivers once they gain a lower privileged permission.

Another related work [6] audits third-party Android phones by comparing them side-by-side to the official Android operating system in order to locate potential security vulnerabilities and design flaws creeping through the vendor customization. The authors

extract pre-installed apps and libraries from a custom Android image, build a matching system from the AOSP, then compare the pre-installed apps and libraries to locate modifications and assess their security. The results of comparing a custom HTC phone (HTC Evo 4G) to a matching AOSP reveals that vendors modified several framework binaries such as: android.policy.dex (security enforcement modules like mandatory screen lock), framework.dex (the core library of the Android framework) and services.dex (hosting a number of built-in system services such as WindowManagerService, PackageMangerService, and ActivityMangerService). The study further demonstrates that the introduced customization might be risky through few discovered cases; a pre-installed app on the HTC test device was found to execute arbitrary commands from a non-authorized IPC channel or the internet, allowing to retrieve sensitive data such as device id and software configuration details. Another detected case reveals an intrusive software (Carrier IQ) collecting different private user data on the test device. My work on Hares (Chapter 4) is different from this work as it focuses on describing a new Android vulnerability and proposing a detection mechanism. My work on harvesting inconsistent configurations in custom Android ROMs (Chapter 5) is similar to this work as it also leverages a comparative approach to locate modifications that could be security critical. This related work though does not look into any configuration discrepancies, but rather, dives into the Java components of the preloaded apps and system libraries.

Another Previous work, conducted by Gallo et al [23], highlights security issues in the Android permission model with regards to Android customization. The authors analyzed five different devices and concluded that serious security issues such as poorer permission control grow sharply with the level of customization.

Dangling pointer protection. Remotely related to my work on Hanging Attributes References vulnerability discussed in Chapter 4 is the prior research on dangling pointers, a memory vulnerability in which a pointer in a program does not point to a valid object [24]. The problem has been studied for decades and can be detected by various tools such as Valgrind [25]. Given the conceptual similarity between this old problem and Hare, the new security risk actually comes from the interconnections among different apps and system components, whose detection and mitigation need to be done across the whole operating system. This poses a new challenge to the system security research.

3.2 Demystifying Android Security

Survey of Android Security. A high-level view of Android security is presented in the earlier research works [26, 27]. The first study [26] unmasks the complexity of Android security, and discusses the two security enforcement mechanisms employed by Android, one applied at system level (e.g., private APIs, permission check enforcement, etc.) and the other at the Inter Component Communication level (e.g., component visibility, permissions, etc.). The work further notes some possible development pitfalls that occur when defining an application's security. The second work [27] aims to better understand Android app attack vectors through a systematic characterization of popular Android apps' security. The authors consider a broad range of concerns including dangerous functionalities and vulnerabilities and proposed a Dalvik decompiler for the analysis. Both works highlight the concern on over-privileged apps and question Android's permission-based security architecture.

A similar study [28] provides another survey of Android security in general. The work first provides a taxonomy of mobile platform attack classes with specific examples (such as Android repackaging attacks, remote execution of payloads, etc.) as each class applies to the Android environment, then proposes mitigations when possible. More recently, SoK [29] systematizes the research work on Android security and privacy in appified platforms. To objectively evaluate and compare the existing different approaches, the authors first came up with a common understanding of several challenges and attack models threatening the Android ecosystem, then created a unified understanding of the attacker's capabilities. Based on this understanding, the authors analyzed the security benefits of different proposed solutions in the Android security literature and performed a systematic comparison with regards to their role in the overall ecosystem. My HareHunting work (Chapter 4) has been categorized by this study as a security consequence of vendor customization/ fragmentation, caused by both modifications carried out by platform developers and device vendors.

Android Permissions. Android permissions are meant to protect critical android resources on the framework layer. Apps can optionally use built-in or custom permissions to protect their own resources (a content provider, a service, an activity or a broadcast receiver). The android literature contains a large amount of research work on Android permissions. Prior work [30] leverages dynamic analysis to demystify android permissions usage, through mapping APIs to their permission requirements. To overcome some limitations of Stowaway [30], PScout [31] proposes a static analysis tool aiming to come up with a complete specification for the Android permission system that lists permission

requirements for every API call. The tool performs a reachability analysis between API calls and permission checks within the entire Android source code. The work [32] is another earlier effort aiming to help developers specify the minimum permission requirement needed for the correct functioning of a given app; the proposed tool relies on a manually constructed API-permission mapping and on the app code's to point out permissions not required by any used API. Compared to Stowaway [30] and PScout [31], this tool involves a less reliable approach to map permissions to APIs; manual parsing of Android APIs documentation which is most often not complete.

Another line of research is devoted to enhancing the permission system. Felt et al. [33] propose making the permission granting mechanism dependent on the permission type requested, e.g., auto-granting non-severe permissions with reversible side-effects, trusted UI for user-initiated or alterable requests, or confirmation dialogs for non-alterable, app-initiated requests that need immediate approval. On the same line, Roesner et al. [34] proposed a concrete realization of trusted UI in the form of access control gadgets that allow a user-driven delegation of permissions to apps whenever such widgets can be effectively integrated into the apps' workflows. A more recent work [35] aims to improve the effectiveness of Android permissions by employing the notion of privacy as contextual integrity. More specifically, the authors propose a new permission requesting model, that would only prompt users when an app accesses sensitive data in a way that defies a user's expectation. The goal of the proposed work is to minimize habituation by only confronting users with necessary security decisions and avoiding to show them permission requests that are either expected, reversible or unconcerning. Liu et al. [36] propose to reduce the list of

permissions that the user faces at application install time and replacing it with a concise list reflecting privacy profiles.

Other research works [37, 38] on Android permissions employs NLP techniques to analyze Android apps descriptions, derive required permissions, and check whether they correspond to the effective permissions set requested within the Android manifest file. Zhang et al [39] take a different approach and generate security-centric app descriptions from analyzing apps' code in an effort to educate Android users about understanding a given apps' functionalities at install time.

Android Linux-layer security. Only limited effort has been made on Android's Linux-layer security whether in terms of investigating possible security threats or providing enhancement to the existing access control. Besides the research work [5] discussed in the earlier section of related works, a prominent work include [40] describing a new side channel for inferring user secrets such as key strokes and browsing history through tracking changes in a target application's memory footprint, exposed by Android's Linux. Another related work is the study [41] inspecting public resources disclosed at both the Android and the Linux layers to measure its impact on sensitive data exposure. The work demonstrates that by monitory various exposed channels (e.g., Linux resources such as process usage data), an app without any permission may acquire sensitive information such as smartphone user's identity, her disease condition, geo-locations and her driving route, from top-of-the-line Android apps. My DroidDiff finding on the Linux GID downgrades through association with non-privileged Android permissions, is related to these works in that it

also describes a new attack vector that can allow third party apps to access critical Linux resources (device drivers) by requesting non-system Android permissions.

Enhancement of Android Security. A lot of research work have been conducted to enhance Android's access control model and improve its security. Quire [42] proposes a security mechanism to address the confused deputy issue through tracking the IPC call chain and allowing an app the choice of operating with the reduced privileges of its callers or exercising its full privilege. The earlier research works [43, 44] propose modifications to the Android framework allowing the user to provide mock data to applications interactively as they are being used. Similarly, the work [45] presents novel privacy controls to protect users' sensitive data through providing shadow data in place of private ones, and through exfiltration blocking.

Another work [46] develops an Android Permission Extension (Apex) framework, a comprehensive policy enforcement mechanism for the Android platform allowing a user to selectively grant permissions to applications as well as impose constraints on the usage of resources. For that purpose, the authors defined the semantics and the policy model used to describe these constraints. They further introduced an extended package installer that enables end users to specify the resource usage constraints. This work is similar to the earlier work Saint [47] which describes a mechanism that enables app developers to define install-time and runtime constraints. However, the proposed Saint framework [47] gives the option of policy specification to the application developers and not to the user as apposed to Apex [46]. Other frameworks such as XManDroid [48] and TrustDroid [49] focus on mediating communication between components in different applications. FlaskDroid [50]

and the SEAndroid project [51] also mediate component interaction as a part of their enforcement. SEAndroid solved the technically complex challenge of porting SELinux-based mandatory access control from the desktop domain to Android.

On a different track, revDroid [52] aims to assess the side effects (such as app crashes) of selective permission granting and revocation proposed by these works [43, 45, 46], through an automatic static analysis that counts unhandled security exception caused by permission revocations.

Another line of research employs tainting capabilities to enhance Android's access control. AppFence [45] provides additional mechanisms to shadow sensitive data and to block unauthorized leakage via the network through fine-grained taint tracking. YAASE [53] encompasses tainting to prevent the confused deputy and privilege escalation attacks. The work [54], on the other hand, employs taint tracking to enforce data-driven usage control in a business environment.

AdDroid [55] and AdSplit [56] provide fine-grained access control at component level. Both of them attempt to restrict untrusted 3rd-party components (i.e., advertisement) inside the app. AdDroid [55] encapsulates the advertising libraries into the Android framework to lift their trust level, which can only be realized if the device vendors reach an agreement with all advertising companies. AdSplit [56], on the other hand, isolate advertisement into a separate activity from the app, such that the advertisement activity is placed beneath the app activity. However, this approach is not likely to be adopted in practice as it imposes a risk over the app's transparency, thus, possibly leading to the Click Jacking attack and its variations. AFrame [57] presents a different isolation approach relying on process ids. The solution places advertisements and its containing app on the

same drawing surface but within different processes. As such, the approach does not suffer from the limitations inherited from using the transparency technique.

3.3 Android Vulnerabilities.

A separate line of research on Android focuses on identifying Android specific vulnerabilities and attacks. My HareHunter paper (Chapter 4) falls into this subcategory as it identifies a new attack model, mainly attributed to non-careful (security oblivious) device customization and that can be exploited to mount high level attacks (e.g. phishing, stealing user info, etc.). However, to the best of my knowledge, never before has anyone investigated the security risks of hanging references: i.e., the parties to be invoked do not exist and can therefore be impersonated by a malicious app.

Android Updates. Few researchers tried to identify vulnerabilities caused by the very fast paced Android app lifecycle and frequent updates released by Google. Thomas et al. [11] collected a corpus of 20400 custom Android devices and demonstrated that there is a significant variability in delivering security updates to Android devices manufactured by different vendors and carriers; leading to unpatched known security vulnerabilities. In fact, the study reveal that 87.7% of the collected Android devices have major security vulnerabilities and are exposed to at least 1 major threat. The research work [58] reveals another class of attacks caused during the Android OS updates through which an attacker can strategically request permissions and other attributes, available in future OS versions, to elevate its privilege once the update takes place. This problem (called Pileup [58]) is most related to my work HareHunter (Chapter 4) in that it is caused by the logic flaws

within the Android upgrade mechanism, which tends to avoid substituting a new app for the existing one with the same attributes such as package names. In a Pileup exploit, new attributes not in use are preempted by a malicious app before an upgrade, while in a Hare attack, the adversary takes advantage of the attributes that do not exist but are still used on a device. My DroidDiff work (Chapter 5) demonstrates other security problems caused by the fast paced API version updates; to catch up with a new version release, vendors might not concentrate on fixing all security vulnerabilities discovered especially in older models, forget to adapt important configuration changes and might even make mistakes through downgrading important security features.

Android and Web. Other researchers focused on uncovering vulnerabilities within specific Android apps in the web landscape. Luo et al. [59] present the first systematic study on the security problems of WebView and discover several attacks revealing a fundamental problem in the weakened TCB and sandbox of the Android's Webview infrastructure. Wang et al. [60] perform another systematic study to understand the unauthorized origin crossing on mobile OSes and bring in light the presence of such vulnerabilities in high profile apps. Recent work [61] has extended the scope to cover code injection attacks on all HTML5-based mobile apps. The conducted attacks mainly target vulnerable apps that utilize the WebView feature provided by Android.

Component Hijacking attacks on Android. Other research work have tackled various vulnerabilities, specific to the Android ecosystem. Security effects of exporting content providers have been studied by Zhou et. al [22], including content leaks and

content pollution. Permission re-delegation [20] attacks depict another consequence of non-intentional public export of interfaces, in which an application with a permission performs a privileged task on behalf of an application without that permission; thus, an attacker can exercise privileged capabilities without acquiring the corresponding Android permissions. Earlier research work [62] has studied unauthorized intent receipt where an attacker can hijack activities and services in case of implicit intents. The work does not directly touch the Hare flaws as it does not require the absence of the legitimate activity/service being referred. Rather, it discusses the cases where multiple recipients are present on the device. In my HareHunter work (Chapter 4), I discuss that even an explicit intent can be hijacked when the legitimate recipient is not in place. I further evaluate the security consequences of hijacking other components such as content providers and permissions.

Other works include evaluating the security risks resulting from design flaws in the push-cloud messaging [63], identifying the risks of Android app uninstallation process [64] and the risks of Android's Clipboard component and sharing mechanism [65]. Two recent studies further examine the crypto misuse in Android apps [66, 67]. These works relate to my DroidDiff research as they are also partially due to developers' mis-configurations of app components or misinterpretation of Android's security protection. Others works include discovered vulnerabilities on Android's flawed design, such as the research [68] exploiting flaws in Android's system server to mount several DoS attacks and the study [69] uncovering the Android root providers and showing that these well-engineered exploits are not well protected, and can be extremely dangerous if exploited.

GUI Security. GUI Security has been extensively studied in the context of Android OS with relationship to its unique design and GUI sub-systems. It has been demonstrated that Android's GUI confidentiality can be breached by embedding UI components from a malicious source [59, 70], through commending adb to take screen shots without user's knowledge [71], via other side channels such as shared memory [72], or reading device sensors [73, 74]. Most recently, the work [75] performs a systematic security evaluation of Android's multitasking and the ActivityManagerService design in depth and discovers a wide open surface of attacks allowing to confuse users about the displayed UI and threaten its confidentiality.

Side channels: Exploiting Unprotected Resources Few research works point out unintended information leaks using motion sensors provided by mobile devices. The feasibility of keystroke inference from nearby keyboards using accelerometers has been shown in [76]. Another work [77] demonstrates the possibility of keystroke inference on a mobile device using accelerometers and mentions the potential of using gyroscope measurements as well, while another study [78] points to the benefits of exploiting the gyroscope compared to other motion sensors on the device to infer keystrokes. The study [79] shows that gyroscopes on smartphones can be used for eaves-dropping on a conversation in the vicinity of the phone and identifying the speakers. Zhou et al. [41] reveal that audio on/off status is a side-channel for location tracking without permissions. SurroundSense [80] demonstrates that ambient sound and light can be used for mobile phone localization via ambience fingerprinting. The recent work [81] shows that by simply reading the phone's aggregate power consumption over a period of a few minutes, an

application can learn information about the user's location. The authors employed machine learning algorithms to accurately infer users' location.

3.4 Android Malware.

Android research is rich in the field of Android malware detection. I place my DroidAPIMiner work (Appendix A) in the context of other approaches to detect malware through leveraging byte-code level information.

One much-studied direction focuses on detecting Android malware based on the permission requirements. Kirin [82] blocks apps that declare risky permission combinations or contain any suspicious action strings used by activities, services or broadcast receivers. Sarma et al. [83] propose different risk signals based on the requested permissions, category as well as requested permissions of apps belonging to the same category. In another work, Sarma et al. [84] employ probabilistic generative models to compute a real risk score of Android apps based on the permissions that they request. Other work for detecting malware through bytecode level information include (AASandbox) [85] which relies on a trial and error approach to identify suspicious patterns in the source code, and DroidRanger [86] which detects Android malware based on the similarities of the requested permissions and the behavioral footprints to different known malware families, formulated through a heuristic based filtering. Compared to these two approaches, my DroidAPIMiner (Appendix A) is more reliable, I do not rely on any heuristics or a trial and error approach. Rather, DroidAPIMiner conducts a thorough frequency analysis of API calls within benign

and malicious apps to extract malware features and employ machine learning to get the most relevant ones.

Another direction of related work relies on system level events to detect possible malicious behavior. Schmidt et al. [87] extract library and system function calls from Android executables and compare them to malware executables to classify apps. Crowdroid [88] collects system call traces of running apps on different Android devices and applies clustering algorithms to detect malicious behavior. More Recently, AsDroid [89] detects stealthy app behaviors by identifying mismatches between API invocations and the text displayed in the GUIs. More similar research [90, 91] to my DroidAPIMiner rely on semantics within the bytecode to detect specific vulnerabilities in Android applications. Potharaju et al. [90] aim to detect plagiarized apps through different detection schemes relying on symbol tables and method-level Abstract Syntactic Tree fingerprints. In [91], Zhou et al. aim to systematically detect and analyze repackaged apps containing additional potential malicious logic in third party Android markets based on fuzzy hashing techniques.

Similar to my proposed detection approach in DroidAPIMiner, Drebin [92] extracts several features from Android applications (e.g., requested permissions, invoked framework APIs) and then applies machine learning techniques to perform classification. I did not rely on Android permission in my analysis, but instead focused only on API calls. I justify this decision with the fact it is quite easy for malicious authors to request more benign permission to defeat any approach relying on them for detection. Zhang et al. [93] extract more sophisticated classification features to fight against malware variants and zero-day malware. More specifically, they extract a weighted contextual API dependency graph as program semantics to construct feature sets and introduce graph similarity metrics to

uncover homogeneous application behaviors while tolerating minor implementation differences.

A different direction for detecting Android malware relies on dynamic analysis. Andromaly [94] continuously monitors various system metrics to detect suspicious activities through applying supervised anomaly detection techniques. In [95], Enck et al. perform dynamic taint analysis to track the flow of private and sensitive data through third party apps, and detect any leakage to remote servers. Portokalidis et al. [96] propose a security model for protecting mobile devices which performs multiple attack detection techniques simultaneously on remote servers hosting an exact replica of the devices.

Other works are more general and aim to detect potentially suspicious behavior such as sensitive data flows and information leaks. Notable works include FlowDroid [97] and DroidSafe [98] which propose precise static taint analyses to detect potentially malicious data flows. Similarly, AppSealer [99], Capper [100], PEG [101] exercised static data flow analysis to identify dangerous code in Android apps. AppContext [102] is a system leveraging supervised machine learning to classify potentially malicious behaviors by taking into account the context in which such behaviors are executed. A more similar approach [103] also shares the same observation with AppContext; that is just looking at behaviors alone is not enough to perform precise classification, and propose a solution to detect logic bombs (i.e, malicious behavior triggered under special conditions).

Lok and Yin [104] present DroidScope, a virtualization based platform for Android malware analysis. It rebuilds both the operating system and Java level semantics, and enables instrumentation of the Dalvik and native instructions. Consequently, DroidScope can be used to understand the behavior of malware both at the native code level as well as

at the interaction with the system. On the same line, AppFence [45] is a dynamic framework implemented as modifications to the Android framework that prevents attacks against user privacy via data shadowing. Other research works, such as Mobile Sandbox [105], CopperDroid [106], Andrubis [107], VetDroid [108] developed tools and techniques to dynamically analyze unknown Android applications for potential malicious behavior. Another research line proposes solutions based on dynamic analysis to perform multi-path execution and dynamic symbolic execution of unknown Android apps [109–111].

4. HARE HUNTING IN THE WILD ANDROID: A STUDY ON THE THREAT OF HANGING ATTRIBUTE REFERENCES

Earlier research work [26] has demonstrated that most of the vendor customization process happens at the level of system apps; that is, vendors often remove specific system apps, add new ones, and modify the identifiers of existing ones. For example, the manufacturer may customize a smartphone OS for a tablet without 3G capability by removing some components, including the messaging and telephony provider apps; however, in the presence of the apps capable of receiving SMS/MMS messages, malware on the tablet could impersonate the missing telephony providers (using its SMS/MMS authorities) to communicate with those apps and their users (e.g., cheating them into believing that their friends are sending them messages from the VoIP channel). Fundamentally, what causes the problem here is the intrinsic interdependent relations between different Android components (apps and framework services), which connect one party to another through references to the latter's attributes such as package, activities, services names, authorities of content providers and permissions: e.g., startActivity called by one app to invoke another's activity (whose name is specified through setClassName). Customizations made to those components, if not well thought-out, could easily break some of such relations, resulting in the references to non-existing attributes (e.g., the authorities of the SMS/MMS providers not on the tablet). We call them hanging attribute references, or simply Hares.

As a side effect of the Android fragmentation, Hares could also be brought in by the third-party developer who designs her app to run on various Android versions, with or without certain service components it utilizes. For example, a reference to the non-existing messaging content provider could also be embedded in a third- party app meant to work on both the smartphone and the tablet. Compared with the customization flaws discovered in the prior research, which are about misconfigurations of Linux-layer device drivers [26], the hanging reference is a framework-layer issue and potentially more pervasive, given the fact that system apps on that layer have always been the focus of a customization [24]. However, such a problem has never been studied, whose security implications, scope and magnitude, therefore, are not clear at all.

In this thesis, we report this new type of vulnerabilities, which we demonstrate are indeed both security-critical and extensive. We show that popular Android devices are riddled with such flaws, which often have serious security implications: when an attribute (e.g., a package/authority/action name) is used on a device but the party defining it has been removed, a malicious app can fill the gap to acquire critical system capabilities, by simply disguising as the owner of the attribute. More specifically, we found that a Hare on Note 8.0 can be exploited to steal the user's voice note and another flaw on Tab S 8.4 allows a malicious app to impersonate the Facelock guard to gain control on the user's login authentication. The popular Tango app contains an unprotected reference to the missing `sms`, which can be leveraged to steal the user's messages. Also through hijacking various packages, activities or missing content providers, the adversary is able to replace Google Email's internal account settings interface, inject activities into LG FileManager and LG CloudHub to steal the user's password, and trick S-Voice into launching a

malicious program whenever the user needs to use the pre-installed voice recorder. Moreover, on Note 3 (phone) and Note 8.0 (tablet), a Hare related to an absent permission can be exploited to steal all the contact information (e.g., email, phone number, etc.) of the device user and even tamper with its content (e.g., changing a friend's phone number, email and URL to those under the adversary's control), when the malicious app does not have the privilege to do so.

To understand the scope and magnitude of the security hazards introduced by Hares, we (along with other collaborators) propose and ran a new tool to automatically evaluate over 97 OS images for Google, Samsung, LG, HTC and Motorola devices. This measurement study shows that unprotected Hares exist on every single device we tested and are completely open to exploits. Also interestingly, I found that though such flaws can be caused by carriers and other parties, apparently they have been primarily introduced by the manufacturers when customizing the same OS to different device models. Further, the problems are still pervasive even on the latest OS versions and phone models, across different manufacturers, indicating that this security risk has yet come to their attentions. These findings point to the gravity of such security hazards and the urgent need to develop effective solutions to address them. We reported the high-profile Hares discovered in our research to Google, Samsung and other related organizations, who all acknowledged the importance of our findings.

Our measurement study was made possible by *Harehunter*, a new tool for automatic detection of the Hare vulnerabilities within system apps. For this purpose, Harehunter first performs a differential analysis, comparing all the attributes defined by the system apps on an Android image with those referred to by them. Any discrepancy between the definitions

and the references reveals a Hare risk. This instance is further evaluated through automatic program analysis to find out whether it is actually protected: e.g., whether a package's signature has been verified before its activity is invoked. If not, then the problem is reported as a likely Hare (*LHare*) case. Running Harehunter on 97 popular device images, we discovered 21557 likely Hares within 3450 vulnerable system apps, which have been documented in a database. This database is utilized by a protecting app we developed, called *HareGuard*, to inspect every newly installed app on these devices, identifying the suspicious ones that attempt to exploit the Hares there, thereby securing the device even before its manufacturer can fix the problems. Our study further evaluated the efficacy and performance of Harehunter and HareGuard, which were both shown to be highly effective. We further discussed the lessons learnt from our study and the effort that needs to be made to avoid similar problems in the development of future systems.

Attribute reference and Android security model. Different Android components (apps or their internal activities, services, content providers, receivers, etc.) are connected together by Inter-Component Communication (ICC), such as *Intent messaging*. An Intent is a message that describes the operations to be performed by the recipient: for example, `startActivity` that triggers an activity (a set of user-interface related operations) associated with an app. The app's package name and activity name can be specified through the Intent, using the method `setPackage`, `setClassName`, `setComponent`, etc. Here the reference from one component to another happens through the latter's attributes, i.e., the package name and activity name. When these attributes have not been set for the communication, the Intent is *implicit* and needs to be resolved by the OS to locate the

recipient capable of handling it. In this case, the sender needs to provide an *action* (e.g., `android.intent.action.Edit` through `setAction`) and other parameters (such as *data*), and the recipient is supposed to declare an *Intent filter* for its component (activity, service, receiver) that matches these parameters in order to get the Intent. Another important Android component is *content provider*, which manages access to an app's databases (structured datasets). To operate on another app's content provider, one must get an URI "`content://authorityname/path`", through which the database table corresponding to the `path` can be read (`query`) and written (e.g., `insert`), under the consent of its owner. In all such ICC communication, once the target of a reference (e.g., package name, activity name, action name and authority name) is not present on the same system, the reference becomes hanging, which can have serious security implications (Section 4.1).

Android protects its information assets through an application-sandbox and permission model, in which every app runs within its own compartment (enforced through the Linux user protection) and can only access sensitive global resources and other app's components (content provider, service, activity, broadcast receiver) with proper permissions. More specifically, the app can specify for each of its components a permission and only process the message or service request from the parties with the permission. For example, a content provider can be guarded with a `readPermission` and a `writePermission`; a broadcast receiver can be configured to get the message only from those with a specific permission. Such permission protection is mostly set statically within an app's manifest file, but it can also be specified programmatically, using the APIs like `checkPermission`. An app that wants to obtain such a permission needs to ask for the user's consent. However, when the party that defines such a permission does not exist on a custom version,

the permission protection becomes hanging: anyone that defines the permission can silently gain the privilege to access protected app components.

Adversary Model. We consider a scenario where a malicious app has been installed on the target device. However, the app does not need to have any suspicious permissions. Actually, in the case of hanging permission protection, it can define the missing permission by its own to launch all kinds of attacks. To deliver the information stolen from the device, the app needs the communication capability. This can be done explicitly by asking for the network permission, which has been requested by almost all apps. Alternatively, the malicious app can utilize other channels, such as browser, to send the data out, as demonstrated in the prior work [112].

4.1 Exploiting Hares

As mentioned earlier, a hanging attribute reference could be an ICC call to a nonexisting package, activity, service (which could be implicitly specified by the action or data filters) or authority of a content provider, or the use of a missing permission to protect an app component (service, activity, broadcast receiver and content provider). In the presence of such a reference, a malicious app that claims its target attribute could gain access to the information assets exposed by the ICC or guarded by the permission. More specifically, when the reference is not *guarded* along the execution path involving the Hare, that is, no validation of the existence and legitimacy of the attribute before using it, the malware that acquires the attribute (e.g. package/authority/permission name) automatically obtains the privilege associated with the attribute and becomes entitled to

get sensitive messages from the sender, utilize its component, etc. Examples of the attacks are presented in the rest of the section.

It is important to note that not every hanging reference is exploitable. It can be protected by verifying the existence of the package that supposes to define it and then verifying its signature (extracted through `getPackageInfo` with flag `GET_SIGNATURE S`), or its application info `FLAG_SYSTEM`, or by checking the current device's model, country code or other properties (e.g. `getProperty`). The presence of such protection was identified in our study through automatic code analysis (Section 4.2.1). On the other hand, if the security check is not in place, a Hare becomes vulnerable to exploits, even though it could still be nontrivial to find the conditions for triggering the code.

In my research, I systematically analyzed 97 Android factory images from major device manufacturers (Google, Samsung, LG, HTC, Motorola), and found 21557 hanging attribute references that are likely to be vulnerable (Section 4.2.2). To understand the security risks they may pose, we built end-to-end attacks on a few Hare instances. Except a small set of them that were discovered manually, which motivated the whole research, most of the Hares, particularly those within pre-installed apps, were detected automatically using Harehunter described in Section 4.2.1. We reported all these security-critical flaws to the manufacturers, including Samsung, LG, Google and HTC. Some of them have already been fixed. Following we elaborate what we learnt about such vulnerabilities and the consequences once exploited. Also, some of the attack apps we built passed the security check of Google Play, while the rest were accepted by other leading app markets like

Amazon Appstore and even Samsung's own app store, which demonstrates that the security risks posed by these vulnerabilities are realistic[1].

4.1.1 Package, Action and Activity Hijacking

Among all the Hares discovered in our research, the hanging references often point to package names and actions. These attributes play an important role in Hare exploits, even when the main targets are other attributes. This is because a missing package can be the owner of absent activities, and actions often need to be specified for receiving the Intent caused by vulnerable references. Moreover, references to nonexisting activities were also found to be pervasive. By exploiting these vulnerabilities, the malware can let a trusted source (a system service or app) invoke a malicious activity, making it look pretty trustworthy to the user. This enables a variety of highly realistic phishing attacks that can lead to disclosure of sensitive data, such as passwords. Following we elaborate a few examples for such Hare flaws and our end-to-end attacks.

A limitation of the exploits on package names is that once the owners of the targeted names are already on Google Play, our attack apps can no longer be uploaded there, as the Play Store does not allow two apps to have the same package names. This restriction, however, is not applied to other attributes. So those not relying on package names can still get into the Store. Also, third-party app stores like Amazon and Samsung typically do not have the target apps of our attacks and therefore the code for hijacking their package names can often be accepted there. Interestingly, we even managed to publish some of the

[1]To avoid causing any damage to those inadvertently downloading our apps, we either removed them as soon as they were approved by the app markets or make sure that they do not send out sensitive user data or perform other actions that could harm the user.

attack apps on Samsung App Store, even though they performed days of manual analysis on our submission.

Stealing voice note. S-Voice is a personal assistant and knowledge navigator service app pre-installed on certain devices (e.g. Note 8.0). One of its features is voice memo: the user can simply say "take memo" or "take note" to activate the functionality and follow the instruction ("please say your note") to record her note. After the note is taken, the app first checks whether another system app `com.vendor.android.app.memo` (`memo` for short) exists, and if so, connects itself to the latter's service by calling `bindService` using an action name specified by its Intent filter. This hands over the note to the `memo` app. In the case that the app is not there, S-Voice looks for another system service to handle the voice note.

We found that S-Voice fails to verify the signature of `memo` when referring to it. As a result, on the device where the app is missing, the references to both its package name and action (through `bindService`) become hanging. A malicious app can then impersonate `memo` using its package/action names to steal the user's voice note. In our research, we built an attack app with the package name of `memo` that defines a service with the action Intent filter `com.vendor.android.intent.action.MEMO_SERVICE`. The app also includes an interface for receiving service requests and data from S-Voice. We ran it on top of Note 8.0, a device that does not have the `memo` app, and successfully stole the voice note recorded from the user. Our attack app was successfully uploaded to Amazon Appstore.

Cheating AOSP keyguard. Prior to 5.0, all AOSP versions after 2.3 support face-based screen unlock, which is done through a system app called *Facelock* (`com.android`

`.facelock`). Once this biometric authentication option is chosen by the user, the Android Keyguard service will bind itself to a Facelock service, enabling the user to use her face and the front camera to unlock her device. More specifically, whenever the security settings fragment within the Settings app is created, Settings app will invoke `isBiometricWeakInstalled` in `LockPatternUtils` framework class to check if the Facelock app is installed. If so, it will add Facelock as an available screen lock option. Later when the user clicks on the option, Settings sends an Intent to Facelock for configuration. After this step is done (which also includes configuring a back-up PIN or Pattern), FaceUnlock is set as the lock screen option. Under the option, whenever the user clicks on a locked phone, Keyguard will bind itself to the face-unlocking service by sending an Intent specifying the action `com.android.internal.policy.IfaceLockInterface` to the Facelock app. The screen is unlocked once Facelock informs Keyguard that the user is authenticated.

A problem here is that on all the AOSP versions prior to 5.0 supporting the FaceUnlock option, the Android framework class `LockPatternUtils` fails to verify the signature of the Facelock app. As a result, on the device model where the app is not present, the reference to its package name becomes hanging and can be exploited by a malicious app. In our research, we installed on Tab S 8.4 an attack app that impersonated `com.android .facelock` along with the required setup activities and unlocking service, and successfully activated the FaceUnlock option. When the option was selected, the attacker app was invoked and consequently set as a phone lock. When the user wished to unlock the screen, the attacker app utilized the action `com.android.internal.policy .IfaceLockInterface` to cheat Keyguard into binding to its service. As a result, the

malware gained full control of the screen unlock process and was able to expose the device to whoever it wanted. This attack poses a particularly serious threat to the multiuser framework provided by Google from Android 4.2, where an attacker purposely installs the malicious Facelock app as a backdoor to other user's accounts. In fact, once installed in the malicious user's account, the app will be immediately enabled on other users accounts as discussed in the prior research [113]. Note that though Lollipop and the later versions no longer offer FaceUnlock, and instead push the support for the functionality to device manufacturers, this security flaw still has a significant impact, given the fact that around 90% of the devices in the market are running the versions below 5.0.

Phantom on Galaxy. Samsung Task Manager is a system app that offers convenient memory management for the user. Through the service, one can monitor which apps are running on the device. In our research, we analyzed the Task Manager and found that it does not display the apps on a white list. Examples of such apps include `com.sec.android.app.wlantest`, `com.kt.iwlan`, `com.sec.imsphone`, etc. Interestingly, those special apps are identified from their package names and no signature verification is in place to check their authenticity. Also, many of them are missing on various devices. The consequence here is that the adversary can build malware exploiting such hanging references, using the authorized apps' names to ensure that his app will not draw attention when it is operating, e.g., recording phone conversation in the background. We implemented such a phantom app on Samsung Note 8.0 and made it disappear from the Task manager. Again our attack app, which masqueraded as a note taker, passed the security check of Amazon Appstore.

Faking Dropbox on LG. LG FileManager is a system app on LG devices that helps the user manage her file system. It also supports the use of Dropbox, which can be opened by clicking on a button with the Dropbox icon. Interestingly, on LG G3 factory image, our analyzer (Section 4.2.1) found that the button actually first tries to launch an activity within `com.vcast.manager`, a Verizon cloud app, and only goes to the Dropbox's web login page once the attempt fails. This program logic could be designed for the devices distributed by Verizon but leaves the reference to the service hanging on those with other carriers and development phones.

In our research, we built an attack app to impersonate `com.vcast.manager` and hijacked the activity pointed to by the hanging reference. Since LG FileManager does not check the target app's signature before starting its activity, it blindly invoked our app whenever the user clicked on the "Dropbox" button. This gives the app an opportunity to show up a fake Dropbox login activity to steal the user's credentials.

Replacing official recorder. S-Voice performs voice recording using a default recorder. There are two such recorders, `com.sec.android.app.voicerecorder` and `com.sec.android.app.voicenote`. What happens is that S-Voice first attempts to use the activity of `voicerecorder` and only when this fails (the app does not exist), it switches to `voicenote`. Again, such a two-choose-one process does not involve proper authentication of the target. This allowed us to construct an attack app impersonating `voicerecorder` app with the activity `VoiceRecorderMain Activity` to control the target of the reference. On Note 8.0, our experiment shows that the attacker's activity was always invoked, even in the

presence of `voicenote`, which enabled it to record sensitive user conversation or perform a phishing attack.

Hulu on watch. WatchON is a popular app that allows its user to view the TV programs in their TV or select movies from the Video-on-Demand service that integrates Hulu, Vudu, popcornflix, etc. Once the user clicks on a Hulu movie, WatchON sends an implicit Intent to launch Hulu's activity. For some movies requiring a HuluPlus account, the user will be redirected to an upgrade activity where she can pay to be upgraded to the HuluPlus status.

The problem here is that the references to the Hulu' activities were found to be hanging in our research: even though WatchON indeed checks whether Hulu exists before sending the implicit Intent, it fails to verify the app's signature. Therefore, we were able to build a malicious app that masqueraded as Hulu and set an Intent filter with action `hulu.intent.action.LAUNCH_VIDEO_ID` to get the upgrade Intent. Through launching a malicious activity, we could cheat the user into entering her login credentials for Hulu. More seriously, when she actually clicked on a paid movie, the malware displayed an upgrade activity, asking for her credit-card information. Since all these activities were triggered by WatchON, the malware is very likely to get what it wants. We successfully uploaded this attack app to Samsung App Store, which analyzed our code both statically and dynamically for days.

4.1.2 Content-Provider Capture

Just like actions and activities, content providers are also extensively used for inter-app and app-framework interactions. Specifically, an app may query another app's content

provider by directly referring to its *authority*, one or more URIs formatted in a Java-style naming convention: e.g., `com.example.provider.imageprovider`. However, just like what happens to other attributes, such a reference (to the authority) can also become hanging, when the related provider is in absence on a device. This opens another avenue for the Hare exploit, when a malicious app strategically defines a content provider to misinform the querier.

Note that unlike package name, duplicated authority names are not forbidden on the Play Store. As a result, all our attack apps were successfully uploaded to Google Play. Following we describe a few attacks on the Hares of this type.

Hijacking Intent invocations. A surprising finding of our research is that a subtle content-provider Hare within Google Email (version 6.3-1218562) allows a malicious app to completely replace its internal account settings with a malicious activity. Specifically, Google Email, the standard email application on every Google phone, lets the user configure different email accounts (Gmail, exchange, etc.) through a Settings interface. To invoke this activity, the app sends an implicit Intent with action `android.intent.action.EDIT` and data `content://ui.email.android.com/settings?account=x`, where x is the email account ID used to inform the account settings activity which email's setting to edit. These two parameters are specified within the account settings activity's Intent filter, as illustrated in the following code snippet:

```
<!-- Account Settings Intent Filters-->
<activity
   android:name=".activity.setup.AccountSettings" android:exported="true">
   <intent-filter>
```

```
      <action android:name="android.intent.action.EDIT"/>
  <category android:name= "android.intent.category.DEFAULT"/>
  <data android:scheme="content"
     android:host="ui.email.android.com"
     android:pathPrefix="/settings"/>
   </intent-filter>
```

This implicit Intent can be received by any app that specifies the above Intent filter for its activity. However, when this happens, Android pops up a window that lists all eligible receivers to let the user select. What we want to do here is to circumvent this protection, making a malicious app the only qualified recipient.

To this end, we analyzed the `data` part of the Intent filter in the code snippet above and checked how the `ActivityManagerSer vice` (`AMS` for short) resolves the Intent sent to this Intent filter. Figure 4.1 depicts the Intent resolution steps in this scenario. If the data's scheme is `content`, `AMS` will try to infer the MIME (Multi-Purpose Internet Mail Extension) of the attached data to identify the recipient that can handle this type: the data type here is supposed to be given by the content provider `ui.email.android.com`. However, this provider does not exist and as a result, the type is typically ignored and the Intent is sent to whoever define the `action.EDIT` and data filter (with `scheme="content"`) without a specified MIME type (as `No` branch in Figure 4.1).

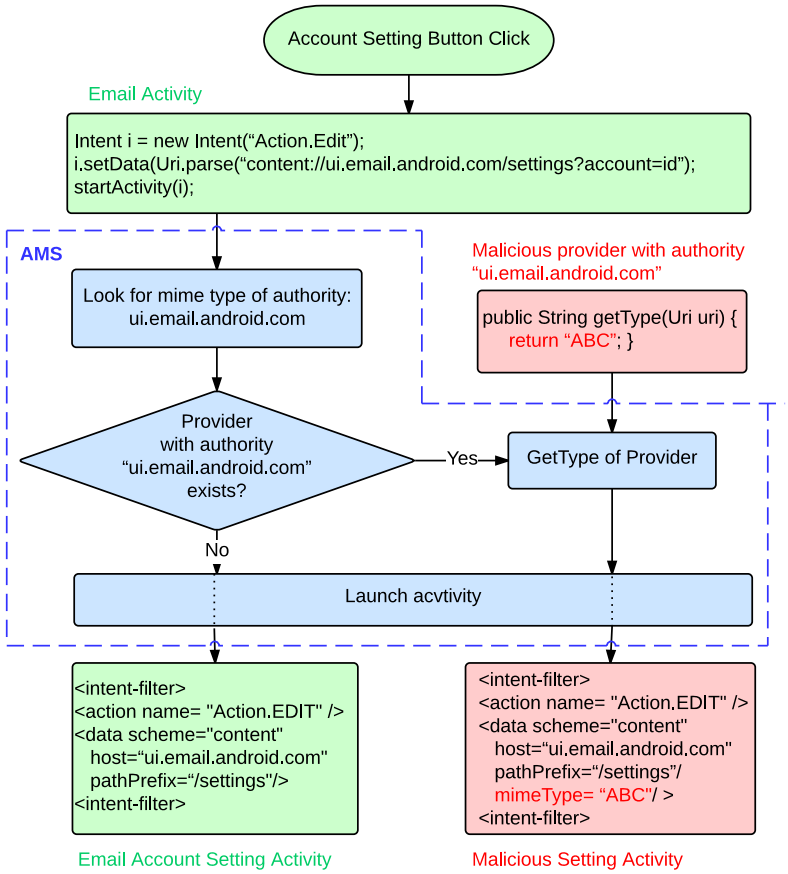


Fig. 4.1.: Exploiting a Hare Authority to Hijack Email Account Settings Activity

The security risk here is that the reference to the content provider is hanging and can be exploited by a malicious app defining that provider. What the malware can do is to name the provider's authority `ui.email.android.com` to receive the query from the `AMS` (the `Yes` branch in Figure 4.1), return a MIME type of its own choice to misinform it, and in the meantime specify this type within its own activity Intent filter, making itself the only eligible app to get the Intent (for invoking the account settings activity). In our research, our attack app took over the content provider and responded to the query from `AMS` with a MIME type `vnd.android.cursor.dir/vnd.example.ABC`. Also, the attacker

defines an Intent filter as illustrated in the next code snippet, by claiming a `mineType` with the type it told the `AMS`.

```
<!-- Malicious Setting Activity Intent Filters-->
<activity android:name=".MaliciousSetting">
   <intent-filter>
  <action android:name= "android.intent.action.EDIT"/>
  <category android:name= "android.intent.category.DEFAULT"/>
  <data android:scheme="content"
     android:host="ui.email.android.com"
     android:pathPrefix="/settings"
     android:mimeType= "vnd.android.cursor.dir/vnd.example.ABC"/>
   </intent-filter>
```

In this way, the Intent from the app went only to the malware, leading the user to a malicious activity that lets her enter her password. We also successfully submitted the app to Google Play, before notifying Google of this security-critical flaw.

Tango in the dark. Tango is a popular cross-platform messaging app, offering audio, video calls over 3G, 4G and Wi-Fi networks. The app has been installed over 100 million times from Google Play. To display SMS messages received, it sets up an Intent filter with the action `android.provider.Telephony.SMS_RECEIVED` to get the Intent that carries the message from the Telephony Manager. When the user sends a message through Tango, the app saves it to `sms`, telephony's content provider.

On a device without Telephony, Tango's reference to its content provider becomes hanging. A malicious app, therefore, can define a content provider using the authority `sms` to get the SMS message the user sends. This can happen when the malware first sends a message, causing the inadvertent user to reply. What can be leveraged here is another vulnerability in Tango: the app does not protect its SMS receiver with the system permission `android.permission.broadcast_sms`, as it is supposed to do. This allows any party broadcasts to the action `SMS_RECEIVED` to inject a fake short message into the app. In our research, we implemented the attack on Tab S 8.4, sending a fake message to Tango and receiving the user's response using the malicious content provider.

LG CloudHub scam. LG CloudHub is a system app that allows managing cloud accounts, uploading data to clouds and accessing it from different devices. By default, the app supports Dropbox and Box, and on various devices can also connect the user to other services, including LG cloud provider. The information about these additional services is kept in a content provider `com.lge.lgaccount.provider`, which LG CloudHub looks up each time when it is invoked.

Interestingly, on some phones, this provider does not exist. A prominent example is LG G3. When this happens, LG CloudHub just displays the default services, Dropbox and Box. However, this makes the reference to the content provider a Hare case and exposes it to the manipulation of a malicious app. Specifically, we implemented an attack app that defined `com.lge.lgaccount.provider` and placed in the content provider an entry for LG Cloud account. This account was then displayed on the LG CloudHub available accounts list. Once it was clicked by the user, the app sent an implicit Intent with action

`com.lge.lgaccount.action.ADD_ACCOUNT`. On the device (G3), no pre-installed apps define the action, which enabled the malware to define the action, claiming that it could handle the Intent. The consequence is that the user's click on the system app (LG CloudHub) triggered a malicious activity that masqueraded as the login page for LG Cloud account, which was used to cheat the user into exposing her password and other credentials.

4.1.3 Permission Seizure

The Hare flaws can also be introduced by permissions, which are defined by system apps and utilized to control the access to various system (e.g., GPS, audio, etc.) or app-defined resources (e.g., content providers, broadcast receivers, etc.). During the OS customization process, the apps that specify the permissions (their original "owners") could be removed. In the meantime, if the resources guarded by these permissions are still there, the uses of the permissions (for protection) become hanging references. To exploit such flaws, the adversary can simply define those missing yet still being utilized permissions to gain access to the resources they protect. This problem was also found to be extensive in our research, present on all 97 factory images we scanned. Making this threat particularly perilous is the fact that Google Play does not check duplicate permissions: all our attack apps were successfully uploaded there. Here we describe two examples.

Getting contacts from S-Voice. The system app S-Voice includes a content provider (`com.vlingo.midas.contacts.content provider`) that maintains the information about the user's contacts, including names, email addresses, telephone number, home addresses, etc. Access to the provider is guarded by a pair of permissions `com.vlingo.midas`

`.contacts.permission.READ` (`READ` for short) and `com.vlingo.midas.contacts.permission.WRITE` (`WRITE`). However, we found that they are not on defined on Galaxy Note 3 (phone) and Note 8.0 (tablet), which opens the door for the exploit.

Specifically, we built an attack app for both devices, which defined the `READ` and `WRITE` permissions. The app was found to be able to successfully read all the contact data from S-Voice and also update its data managed by the content provider at will, e.g., changing the email address, URLs and phone number of a contact, which could lead to information leaks and other consequences (e.g., causing the user to visit the adversary's URL placed in her friend's contact).

Cracking Link. Link is a system app that allows its user to synchronize her data (files, images, audio, video, etc.) across different devices (phone, tablet, laptop, etc.). For this purpose, on a mobile device (phone or tablet), the app uses a content provider `com.mfluent.asp.datamodel.ASPMediaStoreProvider` to maintain the information about such data, together with the geolocations of the user. This provider is protected by `com.mfluent.asp.permission.DB_READ _WRITE` (`DB_READ_WRITE` for short). However, on many factory images, we did not find that the permission has been defined. As a result, the protection here becomes hanging.

We built an attack app in our research that defined the `DB_READ _WRITE` permission. On Galaxy Note 3 and Note 8.0, this app successfully acquired sensitive information from the content provider, including the user's geolocations, all the meta-data of documents, audio and video files (names, directory path, artist, genre, etc.). Also, the malware was able to change the meta-data.

4.2 Detection and Measurement

To better understand Hares and mitigate the security risks they pose, we built a suite of tools in our research, including *Harehunter*, an automatic analyzer that detects Hare flaws from pre-installed apps on factory images, and *HareGuard*, an app that catches the attempts to exploit known hares on a device. Using Harehunter, we also performed a measurement study that inspected 97 factory OS images for popular devices like Galaxy S5, S6, Note 3, 4, 8.0, LG G3, Nexus 7, Moto X, etc. Our study brought to light 21557 likely Hares across these devices, which demonstrates the pervasiveness of such security-critical vulnerabilities. In the rest of the section, we elaborate the design and implementation of these new techniques and our findings.

4.2.1 Harehunter

As mentioned earlier, Harehunter is designed to identify hanging references within system apps and can achieve a high accuracy. We focus on these apps because prior research shows that pre-installed apps are the most intensively customized components across different Android devices [4], and therefore the most likely sources of Hare vulnerabilities. Our manual analysis further indicates that the major portion of Hares indeed come from system apps. On the other hand, framework services may also include hanging references, so do third-party apps (e.g., Tango). Harehunter can be directly applied to find the problems in the third-party apps and extended (by tweaking the pre-processing step) to work on Android services. Following we describe the idea, design of Harehunter and its implementation.

Design. The idea behind our design is simple. For each factory image, we first run a *differential analysis*: extracting all the attributes (package names, actions, activities, services, content providers and permissions) its pre-installed apps define and all the references to the attributes within their code and manifests, and then comparing the references with the definitions. Any discrepancy between these two ends indicates the possible presence of Hares. For example, if a package name is used to start an activity (`startActivity`) or bind a service (`bindService`) but it is not owned by any pre-installed apps on a device, the reference to it is likely to be hanging. On the other hand, such a reference could turn out to be well guarded: for example, before referring to the package, a system app may first check its existence, collect its signature information (e.g., `getPackageInfo` with `GET_SIGNATURE` flag) and verify it against the signature of the authentic app. To detect a truly vulnerable Hare, we have to analyze the code between a potential guard (e.g., functions for signature checking) and a possible hanging reference (e.g., `startActivity`) to find out whether they are indeed related. Only an unprotected reference will be reported as a Hare.

To implement this idea, we designed a system with three key components, *Pre-processor*, *Differ* and *Guard Catcher*, as illustrated in Figure 4.2: Pre-processor extracts app packages from an OS image and converts them into the forms that can be analyzed by follow-up steps; Differ performs the differential analysis and reports possible hanging references; Catcher inspects the APK involving such references to determine whether they have been guarded. In the rest of the section, we describe how these components were built in our research.

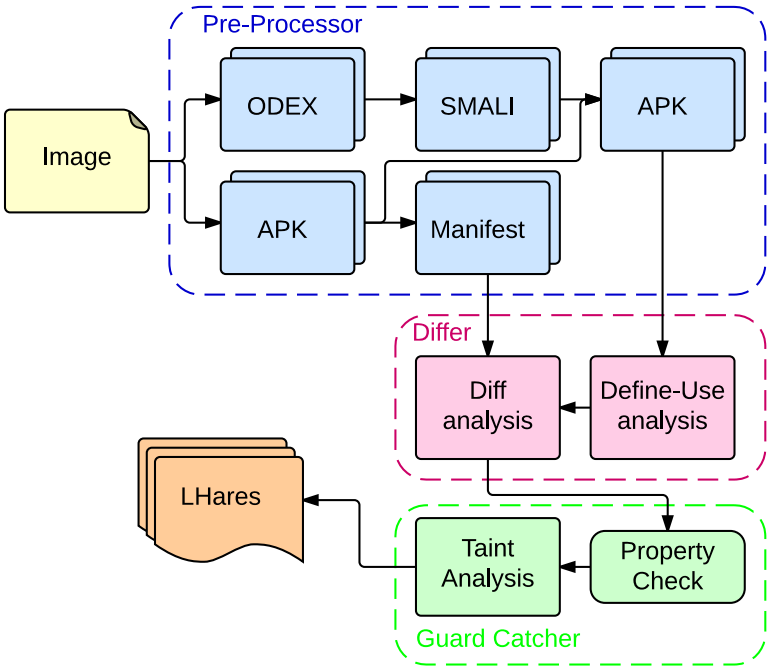


Fig. 4.2.: Design of Harehunter.

Pre-processing. From each factory image, Harehunter first collects all its pre-installed apps, in the forms of APK and ODEX files, and runs `Apktool` to extract each app's manifest file and `Baksmali` to decompile the app into Smali code. For some devices, particularly those with Samsung, a system app's ODEX file is often separated from its APK file, for the purpose of improving its loading time, while *Flowdroid*, the static analyzer we built our system upon, only works on APKs. To address this issue, our pre-processor was implemented to automatically unzip an ODEX file, decompile it and then recompile and compress it, together with its resource files, into a new APK file. Further complicating this process is that for Android 5.0 Lollipop, ODEX files are replaced with OAT files, which include native code. For the app in such a form, Harehunter first

unzips its OAT files and then runs `oat2dex` to convert it to the ODEX formate, enabling the above process to move forward.

Differential analysis. To perform a differential analysis, Differ first searches all extracted, decompiled code and manifest files for the definitions of the targeted attributes. Running an XML parser, our approach can easily collect defined package, actions as well as content providers authorities and permissions from individual apps' manifest file. Note that all these attributes, except the action for receiving broadcast messages, can only be defined within the manifest. Although the action used in an Intent filter for a broadcast receiver can be specified programmatically, it only serves to get a message, not invoke a service or activity, and therefore its absence will not cause a Hare hazard.

Most references to these attributes are within the code, in the forms of various API calls. Specifically, package names and actions are utilized through `startActivity`, `startActivityForResult`, `startService`, etc. The authority name of a content provider appears in various operations on the provider, such as `update`, `query`, `delete` and others. Permissions are claimed in manifests or verified through `checkPermission` and other APIs. To identify these references, Differ first locates the call sites for all related functions from an app's Jimple code (an intermediate representation output by Soot [114]), and then performs a define-use analysis from each call site to recover the targeted attribute names, using the control-flow graph (CFG) constructed by Flowdroid. An issue here is that Flowdroid cannot create a complete CFG, missing quite a few program entry points like `onHandleIntent`. In our implementation, we added back as many entries as we could find, but were still left with some target function calls whose related CFGs could not be built by

Flowdroid. For these calls, our current prototype can only deal with the situation where the attribute names are hardcoded within the related functions.

Guard detection. As mentioned earlier, references to missing attributes are often protected. There are two basic ways for such protection, signature guard or feature guard. Figures 4.3 and 4.4 present the examples for both cases. Signature guard tries to obtain the signature of the package to be invoked, and compare it with what is expected. In the example (Figure 4.3), this check is done through extracting the signature of `com.facebook.katana` through `getPackageInfo` with `GET_SIGNATURES` as a flag and then invoking `compareSignature` to compare it with that of the legitimate Facebook app, before binding to the target app's service (`bindService`). The presence of the authentic package can also ensure the correctness of action and activity names. The other way to protect these attributes is to check the build model of the current device, since only some of them come with certain features (in terms of packages, content providers and others): e.g., input methods, email apps can all be different from builds to builds; SMS/MMS providers may not even exist on a tablet. As an example, Figure 4.4 shows that an app first runs `hasSystemFeature` to check whether the current device supports Google TV (`com.google.android.tv`): if so, it invokes the app `youtube.googletv`, and otherwise, just YouTube.

To detect such protection, Guard Catcher conducts a *taint analysis* through both an app's data flows and control flows, using the functionalities provided by Flowdroid. Specifically, our approach first identifies a set of guard functions like `hasSystemFeature` and `getPackageInfo` with `GET_SIGNATURES` parameter and then attempts to establish

```
public boolean extendAccessToken(Context context, ServiceListener
    servicelistener){
   Intent intent = new Intent();
   try{
  PackageInfo pi = context.getPackageManager().getPackageInfo
      ("com.facebook.katana", PackageManager.GET_SIGNATURES);
  // Compare signature to the legitimate Facebook
  // app Signature
  if (!compareSignatures (pi.signatures[0].toByteArray())){
     return false;
  } else{
     intent.setClassName("com.facebook.katana", "com.facebook.katana.platform.
         TokenRefreshService");
     return context.bindService(intent, new
         TokenRefreshServiceConnection(context, servicelistener), 1);}
   }catch(PackageManager.NameNotFoundException e){
  return false;
   }
}
```

Fig. 4.3.: Signature Based Guard Example

```
private void ViewVideo(Uri uri){
   Intent intent = new Intent("android.intent.action.VIEW", uri);
   if (getPackageManager().hasSystemFeature ("com.google.android.tv")){
      intent.setPackage("com.google.android.youtube.googletv");
   } else{
      intent.setPackage("com.google.android.youtube");
   } startActivity(intent);
}
```

Fig. 4.4.: Feature Based Guard Example

relations between them and the hanging references discovered by the differential analysis, a necessary condition for these references to be protected. For this purpose, the outputs of these guards are set as taint sources and the references (e.g., `startActivity`, `bindService`) are labeled as taint sinks. Flowdroid is run to determine whether the taint can be propagated from the former to the latter. For the sinks that cannot be tainted, they are reported as likely Hares.

Running a full taint analysis (through both explicit and implicit information flows) for every guard and reference pair can be very slow. To make the guard detection more scalable, Catcher takes a multi-step hybrid strategy, combining quick property checks with the taint analysis. Specifically, it first inspects whether a source and its corresponding sink are within the same method. When this happens, in the vast majority of cases, they are related and therefore the reference is considered to be protected. Otherwise, our approach further compares the package name involved in a signature check with that used for a reference. A match found between the pair almost always indicates a protection relation. An example is `com.facebook.katana` within the code snippet in Figure 4.3 that shows up both within `getPackageInfo` and `setClassName`. Only when both checks fail, will the heavyweight taint analysis be used. In our large-scale analysis of factory images (Section 4.2.2), we found that most of the time, the guard for a reference can be discovered in the first two steps.

Evaluation. We evaluated the effectiveness of our implementation in a measurement study, which involves the OS images for 97 popular devices, all together over 24000 system apps. Harehunter reported 21557 likely Hares. From all these Hares, we randomly sampled 250 and manually analyzed their code. Only 37, i.e., 14%, were found to be false detection: that is, falsely treating a guarded reference as a Hare. We further measured the false negative rate of the Guard Catcher by randomly checking likely hanging references reported by Differ and comparing the findings with what was detected by Catcher. In all 250 samples, 46 (19%) were missed by our implementation: i.e., true Hares falsely considered to be guarded. Looking into those false positives and negatives, we found that

Vendor	Images count	System apps count	Avg # of System apps per Image	Countries count	Carriers count	OS versions count
Vendor A	83	21733	261	36	23	10
Vendor B	7	1561	223	1	1	4
Vendor C	1	174	174	1	1	1
Vendor D	4	398	99	1	1	3
Vendor E	2	319	159	2	1	2
Total	97	24185	183	36	23	10

Table 4.1: Android Images Collected

they were all caused by the incomplete call graphs output by FlowDroid. Flowdroid is known to have trouble in dealing with ICC [115] and other issues like missing entry points and incomplete call graphs. When this happens, a taint analysis cannot go through.

4.2.2 A Large-scale Measurement Study

To understand the scope and magnitude of the security hazards caused by Hares, we performed a large-scale measurement study on 97 factory images. The study shows that Hares are indeed pervasive, with a significant impact on the Android ecosystem: over 21557 LHares were discovered and many of them could lead to the consequences such as activity hijacking, data leakage and pollution. Following we report our findings.

OS Image collection. In our research, we collected 97 factory images from Samsung Update [116], Android Revolution [117] and physical devices, which include around 183 apps per image and 24185 all together apps. These images are customized for 49 different phone or tablet models, 36 countries and 23 different carriers. They operate Android versions from 4.0.3 to 5.0.2. The detailed information is presented in Table 4.1. Please note that we are anonymizing vendors upon their request.

Landscape. When analyzing those factory images, we found that about 13% of their pre-installed apps could not be decompiled by `Apktool` or analyzed by `Flowdroid`. Among

Vendor	Hares in Android 4.X		Hares in Android 5.X		Avg Hares per Device	Min Hares per Device	Max Hares per Device
	Hares count	Vulnerable apps count	Hares count	Vulnerable apps count			
Vendor A	19279	3045 (18%)	608	99 (6%)	239	23	598
Vendor B	679	121 (13.3%)	425	85 (15.5%)	157	100	224
Vendor C	N/A	N/A	248	33 (21.5%)	241	248	248
Vendor D	107	31 (12.4%)	8	5 (5%)	29	8	45
Vendor E	187	23 (15.6%)	16	8 (12.1%)	101	16	187
Total	20252	3220 (14.3%)	1305	230 (11.7%)	153	8	598

Table 4.2: Hares Prevalence in System Apps per Vendor

those that could be analyzed, Harehunter discovered all together 21557 flaws (unguarded hanging references) within 3450 vulnerable apps. Note that some of these flaws might occur more than once within the same app, and some of the vulnerable apps show up on multiple devices. Our research reveals that every single image contains a large number of Hare flaws, ranging from 8 to 598. On average, 14.3% of pre-installed apps on 4.X and 11.7% on 5.X were found to be vulnerable. Table 4.2 shows the details.

Also as we can see from the table, the problems are also pervasive across different device manufacturers: both Vendor A and C have a significant portion of their system apps involving hanging references. By comparison, Vendor D has the smallest number of flaws (29) and the lowest ratios (8%) of faulty apps. A possible reason is that the OS images its devices run are the least customized ones, which minimizes the chance for introducing Hares.

Figure A.6 illustrates the distribution of the flaws across different Hare categories. Most problems come from undefined action names. By comparison, a relatively low percentage of permissions were found to be involved in hanging references.

Impacts. The impacts of Hares are significant. In addition to the end-to-end attacks we built (Section 4.1), we also randomly sampled 33 flaws and manually analyzed what could

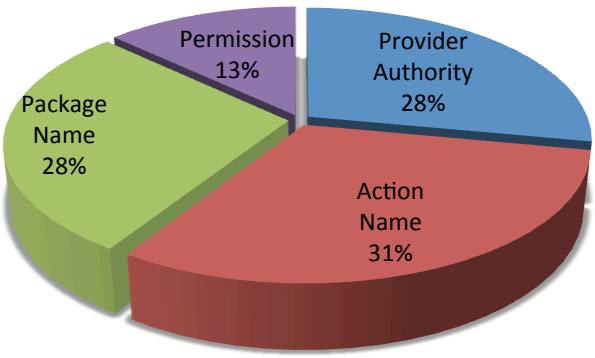


Fig. 4.5.: Distribution of Hares across Different Hare Categories

happen once they were exploited. Note that due to the lack of a large number 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 this type of security flaws that have never been noticed before. The outcomes of our analysis are shown in Table 4.3.

As we can see here, 5 instances of the randomly picked Hares might be exploited to launch similar Phishing attacks as discussed in Section 4.1, due to undefined package and activity names and/or action names for activity Intent filters. One Hare found in the HTC Task App allows redirecting an Intent through exploiting a non-defined content provider used for Intent resolution, just like the GoogleEmail attack. 4 Hares (on the devices such as Note 8.0 and S5) might cause content leakage (notes and browser bookmarks) once malware impersonates undefined content providers, which the victim apps insert data into. 4 instances might expose user's private information when hanging package names are hijacked. Particularly, we found that on Note 8.0, a hanging reference involves an explicit Intent delivered to a nonexisting package. The Intent includes a content URI pointing to

private data (e.g., photos) and also a permission `FLAG_GRANT_URI_PERMIS SION` that enables the recipient to read the data without requesting a permission. As a result, an unauthorized app using the target's package name could gain access to the data.

Also, on LG G3, a hanging reference to a nonexisting content provider might open the door for the adversary to define those providers to contaminate the data synchronized to the user's other devices. Further, our analysis reveals 3 instances that might cause denial-of-service attacks when the adversary creates undefined content providers that victim apps use, and sets their exported flag to false. From the app code, this attack could cause a security exception when the victim app attempts to read or write to these providers. A prominent example is Amazon MP3 app (pre-installed on specific HTC models such as One M8). Once launched, it checks an undefined provider. If a malicious app declares this provider and sets its exported flag to false, Amazon MP3 will never be able to run until the malicious app is uninstalled. Some other Hares may lead to unexpected situations: e.g., an app with a certain package name will not show up in system Task Managers and other apps on LG G3 could not be forced to stop from the LG Settings app.

We also found that Hares in 3 apps might only cause display of dialogs or notifications. Also, there are 6 hares related to missing services whose functionalities we could not figure out. Finally, we did not find any entry points for 4 Hares, which could be dead code.

Responsible parties. We further looked into which parties introduce such flaws and when this happens. For this purpose, we inspected 6 images from Vendor A all running 4.4.2, as described in Table 4.4. The percentage of Hare flaws that are uniquely introduced by these models ranges from 9% to 29%. We further grouped the images into subgroups

Impact	Hare Category	# of Hares
Activity Hijacking	Package and Activity Name	3
Activity Hijacking	Action Name	2
Activity Hijacking	Provider Authority	1
Data Leakage	Provider Authority	4
Data Leakage	Package and Activity Name	1
Data Pollution	Provider Authority	1
D.O.S.	Provider Authority	3
Dialog Popup	Action Name of Activities	3
Others	Package Name	5
Impact Not Clear	Action Name of Services	6
Maybe Dead Code	All Categories	4

Table 4.3: Possible Impact of 33 Randomly Picked Hares

(e.g., phone, tablet) and checked which ones exhibit the highest percentage of common Hare cases. Tablet models have the highest percentage of common Hares 63%, while phone models have the second highest common Hares 56%. The common Hare cases between a tablet and phone device model is at most 38%. So customizing the OS to tablet models or to phone models introduces a lot of Hares. In the meantime, we also compared the flaws found on the same model (Phone 3 running Android 4.4.2)customized for different carriers. The results are in Table 4.5.

Model	# of New Hares Introduced by Model
Tablet 1	106 (27%)
Phone 2	35 (21%)
Phone 3	75 (29%)
Tablet 4	57 (22%)
Tablet 5	22 (9%)
Tablet 6	72 (20%)

Table 4.4: Hare Flaws in Different Vendor A Models Running Android 4.4.2

As we can see from Table 4.5 given a Phone 3 image, its customizations across 6 carriers bring in about 3% to 20% of flaws. Clearly, both manufacturers and carriers cause Hare flaws. However, the former apparently needs to take more responsibility than the latter. Also, most Hares are likely to be introduced during the OS customizations for different device models (phone or tablet).

Country	Carrier	# of Hares Introduced by Carrier
China	China Unicom	51 (20%)
U.S.	AT&T	22 (13%)
Chile	Entel pcs	4 (3%)
Argentina	Movistar	5 (3%)
Brazil	Vivo	5 (3%)
S. Korea	SK Telecom	44 (18%)

Table 4.5: Hares in Phone 3 Running Android 4.4.2 For Different Countries and Carriers

Trend. Figure 4.6 further compares the ratios of vulnerable apps over different OS versions across multiple manufacturers. For Vendor A devices, there is an observable trend that the higher versions (5.0.1 and 5.0.2) contain fewer Hares than the lower ones: the faulty ratio comes from 26% on 4.0.3 down to about 8.2% on 5.0.2. On the other hand, for Vendor B phones, the trend is almost constant: the ratio is 14.3% on 4.2.2 and 15.1% on 5.0.1 . Also, on all these devices, the Hare risks remain significant, which indicates that manufacturers have not yet realized the gravity of this type of vulnerabilities.

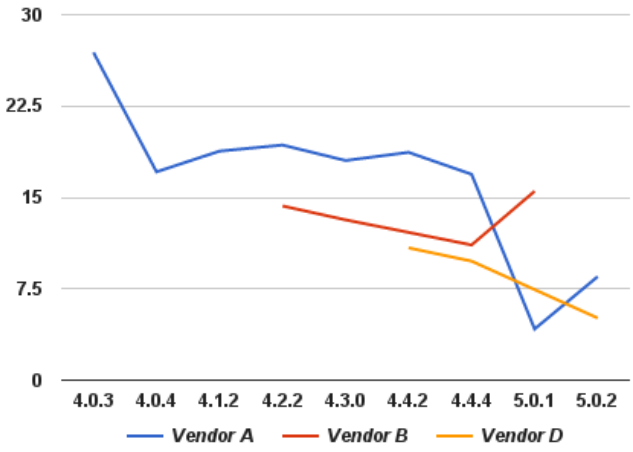


Fig. 4.6.: Ratios of Vulnerable Apps
Across Different OS Versions and Manufacturers

4.2.3 App-level Protection

Motivation and idea. Fundamentally, the Hare flaws can only be fixed by device manufacturers and app developers, who are supposed to either remove the hanging references in their code or put proper security checks in place. However, given the pervasiveness of the problem and its root cause, i.e., the under-regulated Android ecosystem, we believe that they cannot be completely eliminated within a short period of time. Before their complete solution can be implemented (Section 4.3), it is important to help individual Android users protect their systems, in the presence of these flaws. Compared with a framework layer protection, which can only be deployed by manufacturers and carriers, the most practical solution is app-level defense, as all the users need to do is just to install a protecting app from Google Play to get immediate protection against the threats to the vulnerabilities on her system. We found that this can actually be easily done.

In our research, we developed such simple protection, using an app, called *HareGuard*, to scan other third-party apps whenever they are installed to ensure that they are not taking advantage of any known Hare vulnerabilities on a specific device model. HareGuard collects a device's model information and queries a server-side database to acquire all the Hares within the model (which are detected off-line, for example, through Harehunter). Whenever an app is installed, HareGuard immediately checks its manifest file for the package name, activity, action, authority name and permissions it defines, making sure that the app does not intend to hijack any missing attributes. This scanner app is invoked through `startForeground`, running with a notification posted on the Notification Center.

Implementation. Specifically, as soon as HareGuard is installed, it calls `Build` class to collect the device information, including `Build.MANUFACTURER` and `Build.MODEL`, and queries our database for all the Hare flaws on the device. The scanner also utilizes an Intent receiver with actions `android.intent.action.PACKAGE_ADDED` to monitor new app installed and `android.intent.action.PACKAGE_CHANGED` to detect whether an app is updated. For each new or recently updated app, it uses the API `openXmlResourceParser` to open its manifest file and identify all the attributes it defines. These attributes are then compared with a set of hanging references retrieved from our Hare database to detect Hare risks: i.e., defining an attribute associated with a hanging reference. Once a risk is found, HareGuard alarms the user, explaining potential security hazards to her and urging her to make sure that the app indeed comes from a reliable source or simply remove it. To assist the user in this process, the scanner can compare the signature of the app with the one belonging to the authorized party, whenever it exists in the database.

We implemented HareGuard in our research, using a database that documents the findings made by Harehunter when scanning the factory images for popular mobile devices.

Evaluation. Our implementation of HareGuard was found to be effective at detecting all the attack apps we built. We further evaluated its performance, whose impacts on its host system are minimum: the scanner was found to utilize only 4.29 MB memory and consume 0.29% CPU when scanning an app's manifest.

4.3 Discussion

Hares are not just a few isolated, random bugs introduced by implementation lapses. The presence of such flaws implies the weaknesses in Android's design philosophy and its ecosystem. Fundamentally, Android is a complex system, whose components and apps are meant to work together, which leads to highly complicated *interdependent* relations among them. In the meantime, the Android ecosystem is known to be highly diverse and de-centralized: each OS version is customized and re-customized by various parties almost independently and utilized by anyone who can build an app for the version; so far little guidance has been provided to help regulate the customizations and app development, making sure that they respect the existing complicated relations among system components and apps introduced by themselves and other parties (AOSP, manufacturers, carriers, app developers, etc).

In the absence of such guidance and a proper enforcement mechanism, hanging references become inevitable. As evidenced by our research (*the first one on this new category of problems*), indeed Hares are pervasive, existing on every single device we inspected, and also indeed they are security-critical, endangering sensitive user data (e.g., voice memo) and even the proper execution of system apps (e.g., activity injection in Google Email). Even though not every problem reported by Harehunter is exploitable, which depends on the conditions for running vulnerable code, the pervasiveness of such unprotected code is alarming: without deep inspection of individual cases, no one knows whether they can be exploited under certain conditions, leading to unexpected consequences.

Moving forward, I believe that systematic effort needs to be made to eliminate these flaws, and also lessons need to be learnt to avoid the similar pitfall when building other open computing systems. Following are a few thoughts.

Elimination of Hares. To completely eliminate the Hare risks, it is important to have such interdependent relations well documented and make them open to the parties involved in OS customizations and app development. Also, there should be a policy in place that requires that anyone who modifies the OS or builds an app should not create a hanging relation such as referring to a nonexisting attribute, and a mechanism for the policy compliance check. The policy enforcement here can leverage the existing Android compatibility program, which currently still cannot do security check. The challenging part is the collection of the interdependent relations for all known Android versions. Such information is not there yet. Actually, our study shows that the manufacturer seems unaware of the relations on its own device, often breaking them and causing Hares when customizing an Android version to different models. A systematic tool, like Harehunter, is needed to identify such information.

In the meantime, effort should be made to secure each attribute reference. Most importantly here is explicit authentication before a reference. All too often we have seen that references are only protected *implicitly*: e.g., the reference to a system app is secured by the presence of the app on a device, which excludes any other app using the same package name. Such protection is fragile, completely falling apart once the app is removed when the OS is customized for a new device model.

On the other hand, a security check can be more complicated than it appears to be. More specifically, even though references to package names can be directly guarded with a signature check. Other attributes like content providers, actions, etc. can be directly used and their presence on a specific device is often verified by checking the current device model and other features. The correctness of such a check, again, hinges on the knowledge about the components/apps relations across different versions, models, etc., which need to be recovered by Harehunter and other similar tools.

Protection of legacy systems. Before we can even think about how to eliminate Hares in developing future systems and apps, an issue we first need to address is how to secure existing devices, which, as shown in our research, are riddled with different kinds of Hare flaws. The techniques we developed, Harehunter and HareGuard, made a first step toward identification and protection of these vulnerabilities. Particularly, as mentioned earlier, Harehunter can also play a critical role in gathering the interdependent relations to help secure new systems and apps. With its great potentials, our current implementation is still preliminary: it introduced about 14% of false positives and missed 19% of truly vulnerable cases in our study (Section 4.2.1). Most problems are caused by Flowdroid, the static analysis tool that supports our system. It is conceivable that Harehunter will become more effective once a more capable analyzer is used. Also, for device manufacturers who have the source code for all the services and system apps, a tool similar to Harehunter, but working on source code, could be more accurate in detecting the Hare flaws. We expect that these directions will be explored by both the academia and the industry in the near future.

5. HARVESTING INCONSISTENT SECURITY CONFIGURATIONS IN CUSTOM ANDROID ROMS VIA DIFFERENTIAL ANALYSIS

The fragmented Android eco-system brings in several security vulnerabilities when vendors change the functionalities and configurations without a comprehensive understanding of their implications. As discussed in the earlier Chapter 2, previous work has demonstrated some aspects of these changes and the resulting security problems. Wu et al. [4] 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 re-delegation or permission leakage attacks. Our Harehunter work (Chapter 4) reveals a new category of Android vulnerabilities, called Hares, caused by the customization process. Hares occur when a privileged app uses a component that has been removed during customization. A malicious app can "impersonate" the missing component to launch privilege escalation, information leakage and phishing attacks. ADDICTED [5] 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 app components.

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 vendor customization can lead to security problems, what risky configuration changes are often made by vendors, etc.

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

I propose DroidDiff, a tool that detects inconsistent security configurations in a large scale, and that can be employed by vendors to locate risky configurations created unintentionally.

The first challenge that I faced in my systematic study is to identify what configurations are security relevant and are likely to be customized. I call this step *feature extraction*. I start from the Android layered architecture and list access control checks employed at each layer. Then, for each access control check, I rely on Android documentation and my domain knowledge to define corresponding security features. I 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, I have identified five categories of features. DroidDiff then extracts these features from 591

custom Android ROMs that I have collected from multiple sources. This step produces the raw data that will be used for my analysis.

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

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 my discoveries in this chapter 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 investigations. I have identified such risky patterns, and presented results to show how prevalent these patterns are in the customization process. These inconsistencies in security configuration expose devices to potential attacks, but if the vendors understand fully the security 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, my differential analysis can help vendors to identify the inconsistencies introduced during their customization process, so they can question themselves whether they have implemented mechanisms to remedy the risks.

To demonstrate that the identified risky inconsistencies, if introduced by mistakes, can indeed lead to attacks, I have picked a few cases identified from my differential analysis, and designed proof-of-concept attacks on physical devices[1]. We have identified several real attacks from my available devices. To illustrate, I 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 required SEND_SMS permission. Moreover, a detected inconsistency related to permission

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

to Linux GID mapping allows non-privileged apps to access the camera device driver with a normal protection level permission. I have filed security reports about the confirmed vulnerabilities to the corresponding vendors. We strongly believe that vendors, who have source code, more resources, and know more about their systems, can find more attacks themselves from the risky inconsistencies identified from my studies. We also envision that in the future, vendors can use my database and run our DroidDiff on their newly customized images, so they can identify potential risky inconsistencies introduced in their customization process.

5.1 Investigation & Methodology

In this chapter, I investigate Android's security features which are configurable during customization at the level of the framework and preloaded apps. Figure 5.1 depicts my investigation flow. As my 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 my proposed algorithms to find any configuration discrepancies. It should be noted that it is out of my scope to find security features that are wrongly configured on all images, as obviously, they would not be detected through my differential analysis.

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

5.2 Feature Extraction

In this phase, I aim to extract security features that can cause potential vulnerabilities if altered incautiously during the customization process. To systematically locate these security features, I start from the Android layered architecture (Figure 5.2) and study the security enforcement employed at each layer.

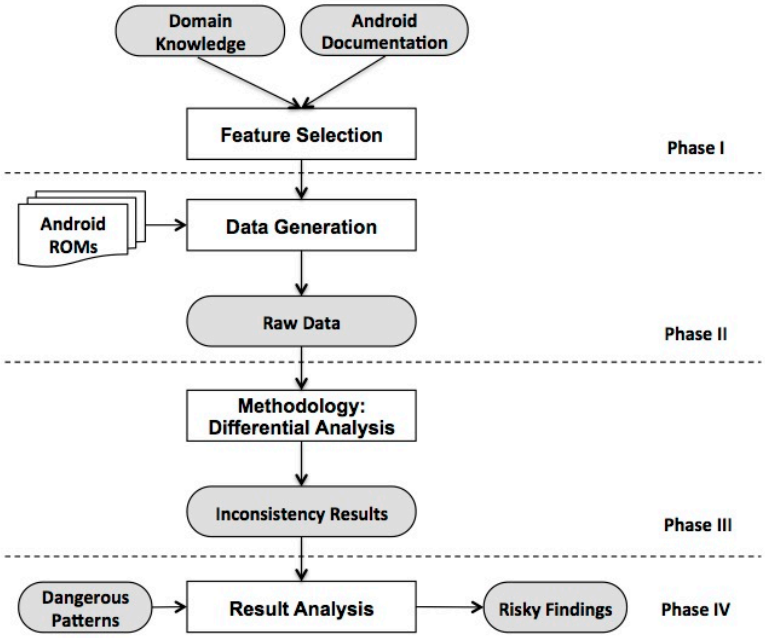


Fig. 5.1.: Investigation Flow

As Figure 5.2 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 5.2, it can make its components private, require the caller to have certain permissions or to belong to a certain process.

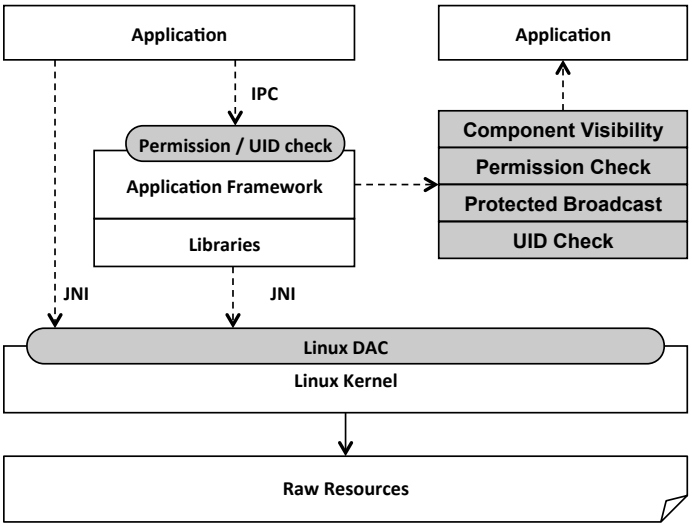


Fig. 5.2.: Android Security Model

Based on Figure 5.2, I summarize the Access Control (AC) checks employed by Android in Table 5.1. I specify the ones whose security features might be altered *statically* during device customization. By *static* modification, I 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, I describe in details each configurable AC check and define its security features based on Android documentation and my domain knowledge. I further justify how inconsistent configurations of these features across custom images can bring in potential security risks. Please note that I do not discuss AC checks based on Package Names as my HareHunter work [118] has covered the effects of customizing them.

Before I proceed, I present some notations that I will be referring to in my analysis. IMG denotes a set of the 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.

5.2.1 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

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

Table 5.1: Security Checks

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 $\in E_P$, I 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}. I 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.

5.2.2 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, I aim to find if the customization parties introduce any modifications to these critical mappings and if so, what damages this might create. More specifically, I 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, I 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, I 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, I define the feature fn_e:

$$fn_e = GIDProtectionLevel(e),\ where:$$

$$GIDProtectionLevel(e) = \min_{\forall p \in P_e} ProtectionLevel(p)$$

5.2.3 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. I hypothesize that because of this, certain protected broadcasts' definitions will be removed as well. I 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 $\in E_{PB}$, I define the following:

$$fn_e = DefineUse(e),$$

Where DefineUse(e) is defined as the following:

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

5.2.4 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. I 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, I consider them as implicitly unexposed.

Formally, for each defined component e $\in E_C$, I 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.

5.2.5 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, I only consider protection mismatches in case of exported components.

We list three cases where a component can be unintentionally exposed on one image, while being protected on other images. The first case is that the permission requirement is removed from the component's declaration. Second, the permission protecting it is of lower privilege compared to other images. Third, the permission protecting the component 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 allow discovering components with conflicting protections, I map their 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 $\in E_c$. I 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, I define P_{read} and P_{write} representing its read and write permissions and extract fn_e for both cases.

5.3 Data Generation

To reveal whether customization parties change the configurations of the mentioned security features, I conduct a large scale differential analysis. I collected 591 Android ROMs from Samsung Updates [116], other sources [119–121], 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 5.2. In total, these images include on average 157 apps per image and 93169 all

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

Table 5.2: Collected Android Images

together apps. To extract the values of the selected security features on each image, I 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, 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 my differential analysis methodologies, discussed in the next section.

5.4 Differential Analysis

In my analysis, I 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 5.3. As shown, I extract 3 security features and their corresponding values from 2 Xiaomi images. For the custom permission e = `MIPUSH_RECEIVE`, the my feature extraction

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

Table 5.3: Security Configurations Map

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, I 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 I 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 I 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, I 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, I designed algorithms that would reveal inconsistencies (if any) caused by each party. Further, for each algorithm, I select my 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. I 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 IMG_{MODEL} denote the candidate image set as the following:

$$IMG_{MODEL} = \{img_1, img_2, ..., img_n\}$$

$$\text{such that } img_i \in IMG_{MODEL} \text{ if } model(img_i) = MODEL$$

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 $\in IMG_{MODEL}$. I define the inconsistency condition under Cross-Version analysis algorithm as follows,

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

The above condition implies that fn_e is inconsistent if there exist two images from the same model 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 I am comparing images of similar criteria across different vendors, I pick candidate image sets running the same OS version (e.g. HTC M8 and Nexus 6 both running 5.0.1). My my intuition here is that if an inconsistency is detected, then the vendor is the responsible party. I formally define the candidate image set as the following:

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

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

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

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

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, I want to uncover any feature fn_e that is inconsistent through different models. For example, I 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, I pick the my candidate image sets running the same OS version, defined as $IMG_{VERSION}$ in the previous example. I further make sure that I am 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 $\in IMG_{VERSION}$. I redefine the inconsistency condition under Cross-Model analysis as follows:

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

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 would like 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 I am comparing images running the same OS version, I pick the my candidate image sets from $IMG_{VERSION}$. I further make sure that I am comparing images running the same model as shown in the following inconsistency condition:

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

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. I pick the candidate image sets from $IMG_{VERSION}$ to make sure that

I am 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.

5.5 Results and Findings

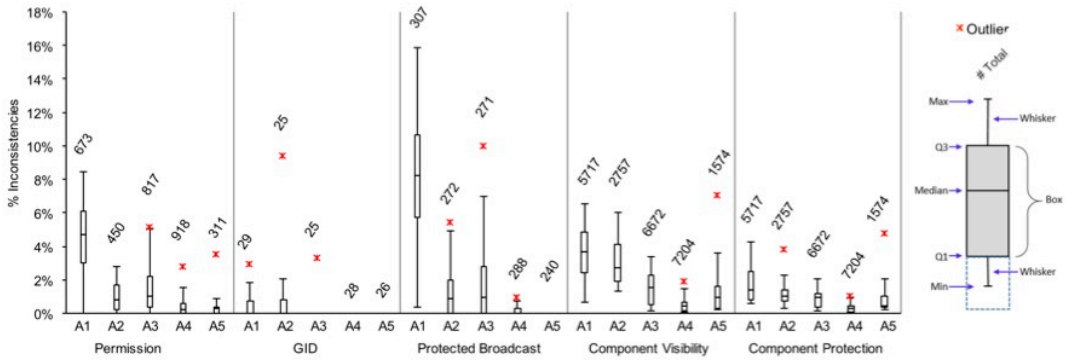


Fig. 5.3.: Overall Inconsistencies Detected
A1: Cross-Version, A2: Cross-Vendor, A3: Cross-Model, A4: Cross-Carrier, A5: Cross-Region

We run DroidDiff to conduct a large-scale differential analysis on our collected images using the aforementioned methodologies. DroidDiff discovered a large number of

discrepancies with regards to our selected features. In this section, I present my results and findings.

5.5.1 Overall Results

Figure 5.3 shows the overall changes detected from my analysis. I plot the average percentage of inconsistencies detected for each feature category (Permission, GID, Protected Broadcasts, Component Visibility, and Component Protection) 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; I 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 5.3 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 5.3, 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 5.5.6.

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. I 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 (potentially very severe) are actually 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.

5.5.2 Permissions Changes Pattern

Protection level mismatch. DroidDiff differential analysis results confirm that Android permissions may hold different protection levels across similar images. As Figure 5.3 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 more insights, I 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`). I have calculated the occurrence of each pattern, and present the results in Figure 5.4. 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 (5.0.1), respectively.
- `com.sec.android.app.sysscope.permission.RUN_ SYSSCOPE` holds `Dangerous` and `Signature` protection on Samsung Note4 and S4(5.0.1), respectively.

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, I found that it is not clear whether developers really intended to use `Normal` as the protection level. I 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, I check its corresponding definitions on other images to see if it has a protection level specified. I then compare the other specification to see it it is `Normal` or not. As Figure 5.5(a) illustrates, on average, 91% of these permissions holding 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. I 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.

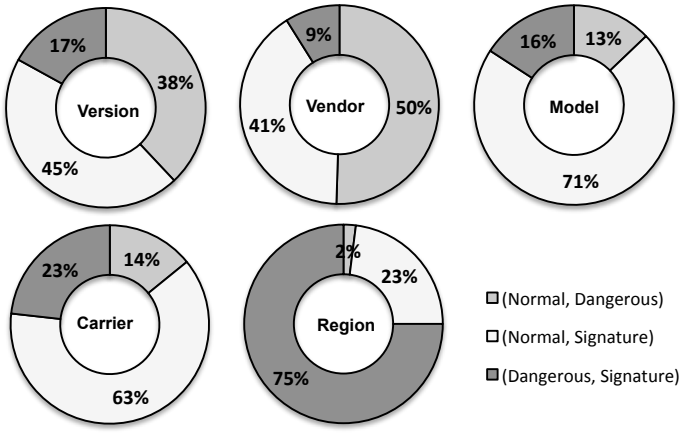
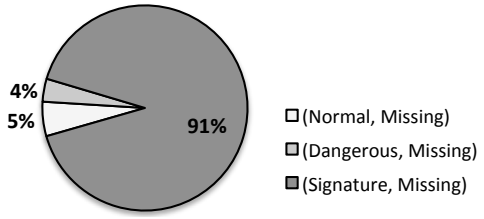
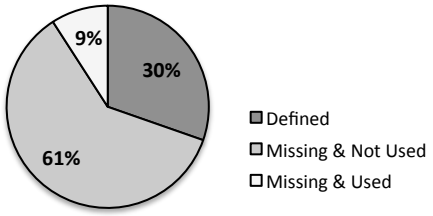


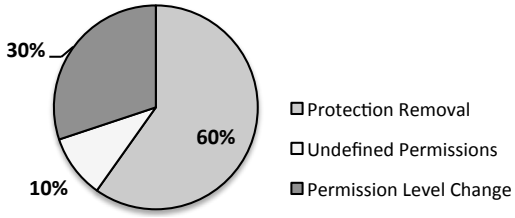
Fig. 5.4.: Protection Level Changes Patterns



(a) Unspecified Protection Level Change Patterns



(b) Protected Broadcast Inconsistencies



(c) Causes of Components Protection Mismatch

Fig. 5.5.: Inconsistency Breakdown

5.5.3 Permission-GID Mapping

By analyzing the differential analysis results of the mappings between GIDs and permissions, I 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 downgrading the protection level of these GIDs. I 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), I 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.

5.5.4 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 5.5(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, my 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 5.3 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.

5.5.5 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.

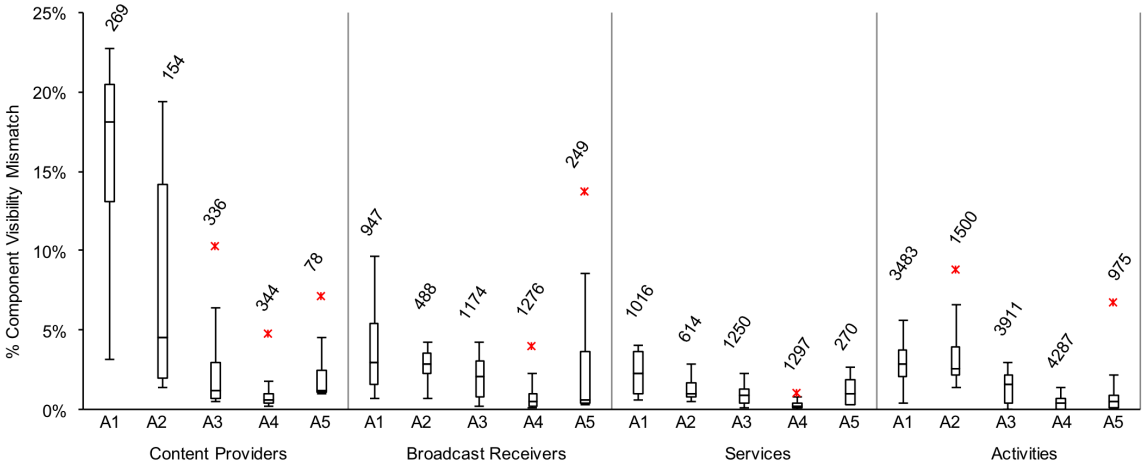


Fig. 5.6.: Breaking Down Components: Visibility Mismatch

As Figure 5.3 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, I break down my findings to activities, services, receivers, and content providers. I plot the results in Figure 5.6. 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 my findings in Figure 5.7. My 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)

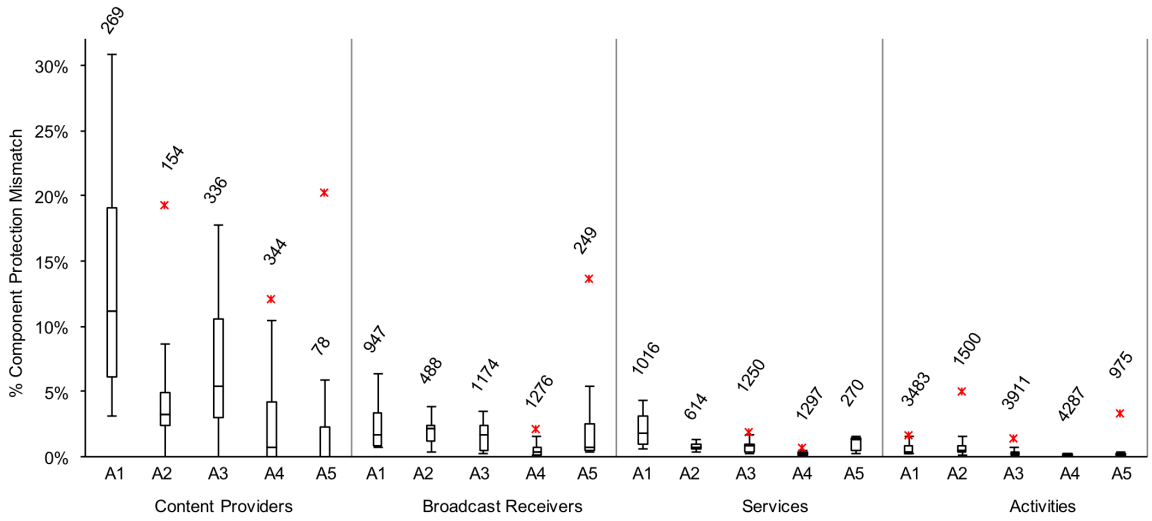


Fig. 5.7.: Breaking Down Components: Permission Protection Mismatch

and Cross-Model (A3) analyses, respectively. I have further analyzed these inconsistent components and categorized the reason behind the discrepancies. As Figure 5.5(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 my 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:

```
<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>
```

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. I have confirmed this dangerous pattern in several preloaded apps and were able to achieve various damages. I filed a bug report about this discovered vulnerability to Android Security team and informed other vendors about it.

5.5.6 Downgrades Through Version Analysis

A dangerous pattern that I 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, my report in Figure 5.8, the percentage of security configuration downgrades out of all detected cases. As Figure 5.8 illustrates, a large number of configurations are indeed

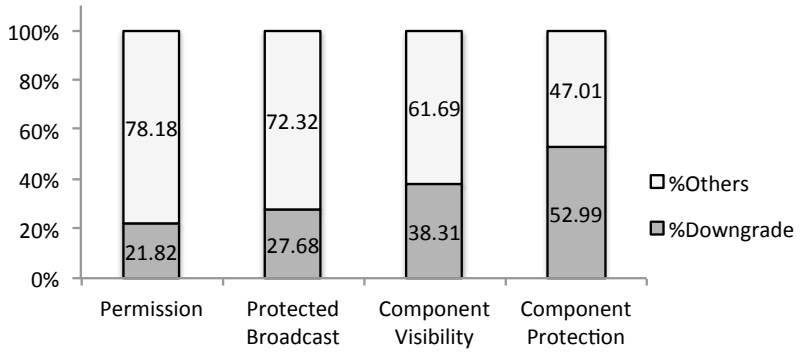


Fig. 5.8.: Percentage of Security Features Downgrades

downgraded. For example, 52% of inconsistent component protection mismatch are actually caused by downgrading the protection.

5.6 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 I 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. I have filed security reports for the confirmed vulnerabilities to the corresponding vendors. I discuss here 6 attacks. At the end of this section, I I 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). I 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. The 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 my analysis reveals a missing definition. I built an attack app that defines the missing permission with a `Normal` protection level. My 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. The 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 5.5.5. 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 5.5.5, using this risky pattern allows a caller app to bypass any restrictions on the first declared broadcast receivers through explicit invocation. I 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. The 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 5.4 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.

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

Table 5.4: Drivers accessible to System GID

Triggering emergency broadcasts without permission. CellBroadcastReceiver is a preloaded Google 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. The 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`. My investigation reveals that this is also due to the duplicate receivers risky pattern (Section 5.5.5). 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. I 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. I 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 I built, I 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 I 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 my analysis are shown in Table 5.5. Please note that I could not assess the impact in 5 cases (heavily obfuscated code), while I confirmed that 2 cases have been hardened via runtime checks.

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

Table 5.5: Impact of Inconsistent Security Configurations

5.7 Limitations

In this section, I discuss some limitations of my 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). My solution pinpoints these possibly *unintentional* risky configurations changes and demands further investigation to confirm whether the change was indeed intentional or not.

Components renaming. My 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.

6. CONCLUSION AND FUTURE WORK

In summary, this dissertation conducts a study of Android customization with regards to security aspects. The objective of this work is to systematically investigate any inconsistencies created as a result of this process and to assess its various security implications. First, my investigation led to the discovery of serious Android security flaws that have never been studied before. The problem, called Hare, has been caused by the conflict in the decentralized, unregulated Android customization process and the complicated interdependencies among different Android apps and components. This work brings to light the significance of this security risk, revealing the damages that can be done on various Android devices, and showing that popular devices are riddled with such flaws. Second, this dissertation makes the first attempt to systematically detect security configuration changes introduced by Android customization. I 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, I were able to locate critical security changes that might have been unintentionally introduced during the customization. My systematic comparison shows that indeed, customization parties change several configurations leading to severe vulnerabilities such as private data exposure and privilege escalation.

As future work, I propose to enhance DroidDiff to detect risky inconsistencies more accurately; I plan to detect the cases where a component's implementation has been

hardened via runtime checks (invocation of specific APIs). To detect other cases that might be missed because of components' renaming, I intend to calculate the similarity between components and then check the security configurations of any two similar components. I believe that such improvements to DroidDiff would help reduce the number of false positives and pinpoint risky configurations more accurately.

Besides, I plan to employ DroidDiff differential analysis results to predict the correct security configuration of a given misconfigured feature. If the majority of security features share the same configuration, then inconsistent components should be most probably configured similarly on the victim images.

Finally, I plan to deploy DroidDiff to be used by vendors to check the configurations of various security features on a given image. DroidDiff will extract those features from the image, and compare them to my collected configurations of other images. Then, DroidDiff would flag inconsistent ones to be further investigated by the vendors who have the source code and devices to check their effects.

A. DROIDAPIMINER: MINING API-LEVEL FEATURES FOR ROBUST MALWARE DETECTION IN ANDROID

As Android mobile devices are becoming increasingly popular, they are becoming a target of malware authors. To protect mobile users from the severe threats of Android malwares, different solutions have been proposed. Several systems have been proposed based on Android permission system. In [122], if an app requests a specific or a combination of critical permissions, a risk signal will be raised. In [83], several risk signals have been proposed depending on an app's requested permissions, its category, as well as the requested permissions from apps belonging to the same category. In [123], different risk scoring schemes have been designed using probabilistic generative models. However, the permission-based warning mechanisms fall short for several reasons:

- The existence of a certain permission in the app manifest file does not necessarily mean that it is actually used within the code. According to [124–126], a large percentage of Android apps are over-privileged.
- A large number of requested permissions, specially the critical ones, are actually not used within the application's code itself, but rather are required by the advertisement packages.
- Malware can perform malicious behavior without any permission [127].

Another direction to detect malicious activities in Android apps relies on the semantic information within the application bytecode. CHEX [128] statically vets Android apps for component hijacking vulnerabilities through performing data flow analysis and conducting reachability tests on the generated system dependency graphs to detect potential hijack enabling flows. Similarly, Woodpecker [127] exposes capability leaks through using data flow analysis and exploring the reachability of a dangerous permission from a non-protected interface. While these approaches are effective in detecting the particular vulnerabilities that they target, they cannot be generalized to detect other malicious activities. DroidRanger [86], on the other hand, combines permission-based behavioral footprints and a heuristic based filtering scheme to detect malicious apps.

In this work, I aim to overcome the shortcomings of the permission-based warning mechanisms and build a robust and lightweight classifier for Android apps that could be used for malware detection. To select the best features that distinguish between malware from benign apps, I rely on API level information within the bytecode since it conveys substantial semantics about the apps behavior. More specifically, I focus on critical API calls, their package level information, as well as their parameters.

Instead of following a heuristic based approach for identifying critical features for malware functioning, I have analyzed a large corpus of benign and malware samples, generated the set of APIs used within each app, and conducted a frequency analysis to list out the ones which are more frequent in the malware than in the benign set. Furthermore, for certain critical APIs which were frequent in both sample sets, I have conducted a simple data flow analysis on the malware APK samples to identify potentially dangerous inputs. I generated a list of frequently used parameters, thoroughly examined them to filter

out the dangerous ones and flagged all apps that request them. To perform API level feature extraction and data flow analysis, I have developed a tool called DroidAPIMiner built upon Androguard [129] reverse engineering tool. I use RapidMiner [130] to build the classification models.

In summary, the contributions of this work are as follows:

- I introduce a robust and efficient approach for describing Android malware that relies on the API, package, and parameter level information.
- Based on the identified feature set of Android malware, I provide valuable insights about malware behavior at API-level.
- I produce and evaluate different classifiers for Android apps. Our testing shows that some of them achieve a high accuracy and low false positive rate compared to the permission-based classifiers. In fact, KNN achieves a 99% accuracy and 2.2% false positive rate.

A.1 Approach Overview

In our work, I I follow a generic data mining approach that aims to build a classifier for Android apps. The classifier should be able to automatically learn to identify complex malware patterns and make smart decisions based on that. The classifier should also be able to generalize from the input set to correctly predict an accurate class of given new apps. As depicted in Fig. A.1, our approach is divided into three phases: feature extraction, feature refinement, and models learning and generation.

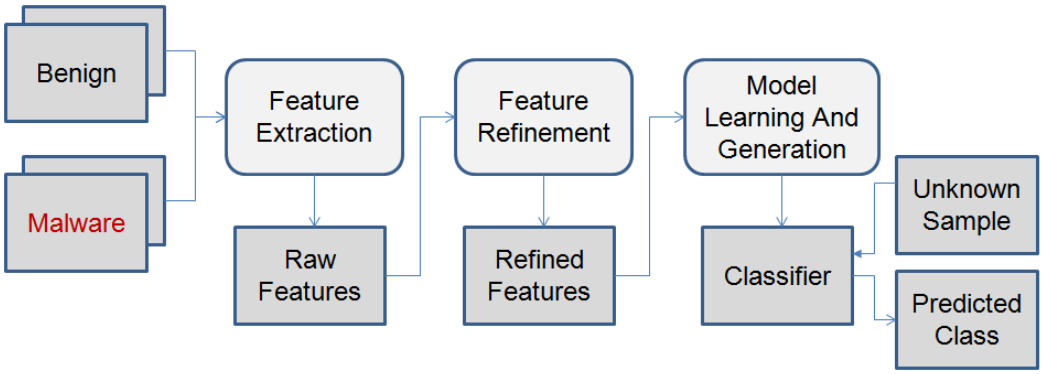


Fig. A.1.: Our Approach

During the feature extraction phase, I statically examine the collected benign and malware APK samples to determine and extract the necessary features for malware to function. In selecting the feature set, I focus on some semantic information encapsulated within the bytecode of apps. More specifically, I extract API calls and their package level information. Besides, I extract the requested permissions of the apps for the generation of the baseline model.

During the feature refinement phase, I remove the API calls that are exclusively invoked by third-party packages such as advertisement packages. I reduce our feature set further to include only those APIs whose support in the malware set is significantly higher than in the benign set. For those APIs which were frequent in the two sets, I perform data flow analysis to recover their parameter values and select only the APIs that invoke dangerous values. Subsequently, for each APK file, I generate a set of feature vectors along with associated class labels, i.e. malware or benign. For the last two steps, I have implemented DroidAPIMiner, a python program that import libraries from Androguard static analysis tool for Android apps [129]. Section 3 will be dedicated to discuss in more

details how I conduct feature extraction and refinement. I discuss in Section 4 some of the insights that I have gained based on the identified features.

During the model learning and generation phase, I feed the representative vectors to standard classification algorithms that build the models by learning from them. I have generated 4 different classifiers: ID5 DT [131], C4.5 DT [132], KNN [133] and SVM [134]. I test the generated classifiers to estimate the accuracy using split validation. Two thirds of the data set are randomly selected for training and the rest one third is dedicated for testing. For this step, I use RapidMiner [130] to generate the classification models and evaluate them. In Section 5, I perform the classification and evaluate the models.

A.2 Feature Extraction and Refinement

In this section, I aim to systematically determine and extract necessary features for malware functioning. Android app's bytecode contains information that could be used to describe its behavior. From the bytecode, I can retrieve information ranging from coarse-grained levels as packages to fine-grained levels as instructions. I do not perform sophisticated program analysis because it is computationally expensive. Rather, I focus on extracting package and API level information since they clearly capture the app's behavior. More specifically, I consider class name, method name, and some parameters of the callee and the package name of the caller, which I will describe in the next subsections.

A.2.1 Extraction of Dangerous APIs

Contrary to previous work, I do not follow a heuristic-based approach to identify dangerous APIs for malware functioning. Instead, I aim to reliably identify the major APIs that malware invoke by statically analyzing our samples.

Effectively, I have statically analyzed a large set of malware and benign apps and generated a list of distinct API calls within each set. A distinct API refers to a distinct combination of Class Name, Method Name, and Descriptor. I then conduct a frequency analysis to select those APIs which are more used in the malware than in the benign set. I further refine the API list to include only those with a usage difference higher or equal to a certain threshold.

A.2.2 Extraction of Package Level Information

Most of Android apps contain one or more third-party packages (according to our analysis, 71 % of the benign apps contain at least one advertisement package). These packages often exhibit some suspicious behavior. For instance, many ads use encryption to hinder their removal. Also, `getCellLocation()` and `getDeviceId()` methods are often called by ad kits for users' identification and tracking purposes. I aim to identify at what package level a certain API is invoked. To achieve this goal, I have performed the following tasks:

- **Extract advertisement and similar packages:** Using Androguard, I generate all distinct packages invoked within each APK in our collected sample. I remove from the generated packages names all common packages such as Android specific

packages, Java packages, etc. I inspect the remaining items and compile a list of advertisement, web tracking, web analysis and application ranking packages. In total, I have identified around 412 distinct advertisement and similar packages. Some commonly used advertisement packages are: Admob, Flurry, Millennialmedia, Inmobi, Adwhirl, Adfonic, Adcenix, etc.

- **Identify calling packages:** I check at what package a certain API is called. In other words, I distinguish if an API is invoked only by a third-party package, only by the application specific packages, or by both. I white-list any APIs that are exclusively invoked by third-party packages.

A.2.3 Extraction of APIs Parameters

Certain frequent APIs in the malware set did not yield to a high support difference between the malware and the benign sample as they were also common in the benign sample. For example, some methods within string manipulation and IO classes are almost as frequent in the malicious set as in the benign set. To increase this difference, I have performed data flow analysis on these specific APIs in order to recover the parameters values that have been passed to them through inspecting the registers invoked.

Classes	Methods	Parameter Category
Intent IntentFilters	setFlags, addFlags, setDataAndType, putExtra, init	Flag is either: CALL, CONNECTIVITY, SEND, SENDTO, or BLUETOOTH
ContentResolver	query, insert, update..	URI is either: Content://sms-mms, Content://telephony, Content://calendar, Content://browser/bookmarks, Content://calllog, Content://mail, or Content://downlaods
DataInputStream BufferedReader DataOutputStream DataOutputStream	init, writeBytes...	Reads from process Reads from connection Uses SU command
InetSocketAddress	init	parameter IP is explicit or port is 80
File Stream StringBuilder String StringBuffer	init, write, append, indexOf, Substring	Dangerous Command such as: su, ls, loadjar, grep, /sh, /bin, pm install, /dev/net, insmod, rm, mount, root, /system, stdout, reboot, killall, chmod, stderr Accesses external storage or cache Contains either: An identifier (e.g. Imei), an executable file(e.g. .exe, .sh), a compressed file (e.g. jar, zip), a unicode string, an sql query, a reflection string, or a url

Table A.1: Categorization of Parameters to Frequently Used Malware APIs

Based on our initial investigation, these APIs generated distinct parameters which resulted in a big number of features. To reduce the parameter feature set, I have categorized the parameters based on different criteria. Table A.1 includes the APIs on which I have performed the data flow analysis along with the criteria that I have adopted to categorize their input parameters.

A.3 Insights in API-Level Malware Behavior

Based on the API level analysis, I have identified the top APIs that Android malwares invoke. Fig.A.2 shows the top 20 APIs that produce the highest difference of usage between malware and benign apps. As illustrated, I get a better difference after filtering out third-party packages. For example, the method `init` in `Java.Util.TimerTask` initially produced 14% usage difference between the two sets. This difference increased to 28% after whitelisting this API in third-party packages since it is mainly invoked by them in the benign sample.

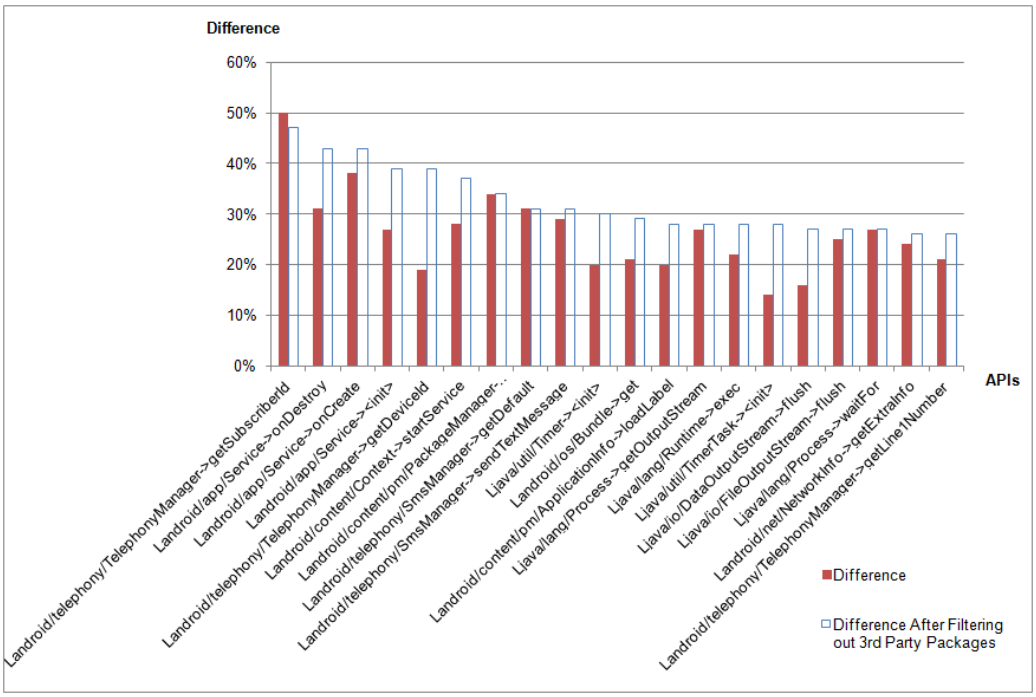


Fig. A.2.: Top 20 APIs with the Highest Difference Between Malware and Benign Apps

We discuss here some of the top commonly used malware features that our study generated after refining the initial feature set. To help understand malware behavior and gain more insight into what resources are accessed and what actions are performed, I classify the APIs by the type of requested resources and utilities. At the end of the section, I present the data flow analysis results.

A.3.1 Application-specific resources APIs

Content Resolver: This class provides access to content providers. It processes requests (CRUD operations) by directing them to the appropriate content provider. The most frequent methods used in this class by malware are `insert()`, `delete()` and `query()`.

This latter can be invoked to grab sensitive information from content providers of other apps if they are not protected by permissions. As stated in [135], some vendor pre-installed apps have implicitly exported content providers which allowed other apps to successfully obtain sensitive information from them without acquiring the necessary permissions.

Context: Context class provides global application information such as its specific assets, classes, and resources. `startService()` is very frequently used methods within this class with a support of more than 70% in malware and less than 34% in benign ones. This API can be invoked to start a given service in the background without interacting with the user. `getFilesDir()` and `openFileOuput()` are other frequent APIs in this class that malwares call to create files and get their absolute paths. `getApplicationInfo()` is often used by malwares for obtaining various information about the app such as whether it's debuggable, installed on external storage, holds factory test flag, etc.

Intents: Intents allow launching other activities and services and interacting with the phone's hardware. The most frequent APIs used by malwares within Intents are `setDataAndType()`, `setFlags()` and `addFlags()`. `setDataAndType()` allows setting the URI path for the intent data with an explicit MIME data type. As stated in the official documentation of Android [136], this method should "very rarely be used" since it allows to override the ordinary inferred MIME type of data of a newly specified MIME type. `setFlags()` and `addFlags()` are used to set the old flags or add new ones to the intent to specify how it should be handled. Depending on the parameter flag to these APIs, malware controls the associated component such as running it with foreground priority.

A.3.2 Android framework resources APIs

ActivityManager: This class allows interacting with other activities running in the system. The method `getRunningServices()` is often invoked by malware to inquire whether a certain service (like Anti-virus) is currently executing. `getMemoryInfo()` is also frequently invoked by malware and might be used to check how close the system to have no enough memory for other background process and thus needing to start killing other processes. `restartPackage()` is often invoked by malware to kill other apps' services. According to Android's documentation [137], the original behavior of this method is no longer available to apps as it "allows them to break other applications by removing their alarms, stopping their services, etc".

PackageManager: This class contains information about the application packages installed on the device. Malicious apps call `getInstalledPackages()` to scan the system against a list of known anti-virus and take an appropriate action based on that (e.g. remain dormant, kill the anti-virus process, etc.) .

Telephony/ SmsManager and telephony/ gsm/ SmsManager: These classes allows managing various SMS operations. Malware authors invoke many methods within theses classes. `sendTextMessage()` is very frequently used by malwares authors to send sms messages to premium rate numbers without the user's consent and thus incur financial losses. Examples of SMS Trojans include malware belonging to the following families: SpyEye, OpFake, Gemini, etc.

TelephonyManager: This class retrieves various information about telephony services on the device. The most frequently used APIs by malwares are: `getSubscriberId()`,

`getDeviceId()`, `getLine1Number()`, `getSimSerialNumber ()`, `getNetworkOperator()`, and `getCellLocation()`. Malware authors collect these private data and send it to remote servers to build users profiles and track them. As illustrated in Fig. A.2, `getSubsriberId()` is the mostly used API by our malware sample.

A.3.3 DVM related resources APIs

DexClassLoader: This class allows loading classes from external .jar and .apk files containing a classes.dex. `loadClass()` is one of the most frequently invoked APIs by malware and is used to execute code not installed as part of the app and consequently evade malware detection techniques that rely on static analysis.

Runtime and System: Runtime class allows apps to interact with the environment in which they are running. Malware invokes `Runtime.getRuntime.exec ()` method to execute dangerous Linux commands along with the supplied arguments in a newly spawned native process and thus avoid the normal execution lifecycle of the program. System class provides system related facilities such as standard input, output and error output streams. `loadLibrary()`dynamically loads native libraries and can be used maliciously through running native code exploiting some known system vulnerabilities.

A.3.4 System resources APIs

ConnectivityManager, NetworkInfo, and WifiManage: These classes provide network related functionalities such as answering queries about different connections (Wifi, GPRS, UMTS) and network interfaces. Android malware calls APIs within

ConnectivityManager class (`getNetworkInfo()`), NetworkInfo (`getExtraInfo()`, `getTypeName()`, `isConnected()`, `getState()`), and within WifiManager (`setWifiEnabled()` and `getWifiState()`) to establish a network connection and interact with malicious remote servers.

HttpURLConnection and Sockets: APIs within these classes are used to send and receive data over the web and establish communication with remote servers. The most frequent APIs used by malwares in HttpURLConnection are `setRequestMethod()`, `getInputStream()`, and `getOutputStream()` which manage transferring data between the malware apps and the malicious servers. Similarly, malware applications often invoke `getInputStream()` and `getOutputStream()` in Socket class for the same purpose. I have also noted a heavy use of InetSocketAddress which implements an IP socket address given an IP address and a port number.

OS package: A lot of frequently used APIs in malware belong to OS package which allows message passing, ipc services, process and threads management. `sendMessage()` method in os.Handler class inserts messages into message queues of different executing threads, while `obtainMessage()` retrieves messages from the message queues. Malware authors often invoke `myPid()` and `killProcess()` in Process class to request killing processes based on a given pid. However, the kernel will impose restrictions on which processes an application can actually kill [138]; only apps and packages sharing common UIDs can actually kill each other. Unfortunately, these restrictions will not prevent Android malware from killing processes beyond their scope once they can root the device.

IO Package: IO package provides IO processing services such as reading and writing to streams, files, internal memory buffers, etc. Malwares invoke APIs within

IO.DataOutputStream (such as `writeBytes()`) to write data and upload files through a URL connection. Similarly, they call APIs in IO.DataInputStream (such as `readLines()`, `available()`) to read and download malicious payloads from a certain URL connection. Methods within IO.FileOutputStream (such as `write()`) are used to write the malicious content downloaded from a remote server to local files. `mkdir()`, `delete()`, `exists()` and `ListFiles()` are other used APIs in IO.File by malware for file management.

A.3.5 Utilities APIs

String, StringBuilder and StringBuffer: These classes provide an interface for creating and manipulating strings. Malware heavily call `substring()`, `indexOf()`, `getBytes()`, `valueOf()`, `replaceAll()`, and `Append()`. These methods can be used for code obfuscation, construction of payloads to be sent to servers, and evasion of static malware detection techniques through dynamically creating URLs, parameters to reflection APIs, and dangerous Linux commands.

Timer: Timers facilitate scheduling one-shot or recurring tasks for future execution. Malware can invoke APIs within this class (such as `schedule()` and `cancel()`) to avoid dynamic analysis by remaining dormant until a fixed date is reached, or until a specific event has been fired.

ZipInputStream: This class allows decompressing data from an InputStream ZIP archive. Malwares rely on methods in this class to decompress and read data from compressed files (.jar, .apk, .zip) downloaded during execution or originally attached to the

app. Commonly used APIs by malware in this class are `read()`, `close()`, `getNextEntry()` and `closeEntry()`.

Crypto: This package serves as an interface for implementing cryptographic operations such as encryption, decryption, and key agreement. Methods within Crypto.Cipher such as `getInstance()` and `doFinal()` transform a given input to an encrypted or decrypted format while `Crypto.spec.DESKeySpec()` allows specifying a DES key. These methods can be used for code obfuscation and avoiding static detection through encrypting root exploits, SMS payloads, targeted premium SMS numbers, and URLs to remote malicious servers.

w3c.dom: This package provides the official w3c Java interfaces for the Document Object Model (DOM), which is used in apps for XML document processing. Malwares use several APIs in w3c.dom such as `getDocumentElement ()`, `getElementByTagName()`, and `getAttribute()` to parse XML files. XML can be used by malwares to establish bot communication, encode data, and process local configuration files.

A.3.6 Parameters Features:

Based on the data flow analysis that I have conducted, I obtained the frequent parameters (categorized as discussed in Table A.1) that are used by malwares applications more often than the benign ones in certain API invocations. Table A.2 depicts some of the top invoked parameters types that yield to the highest support difference between the malware and benign sample.

From the data flow analysis results depicted in Table A.2, I can gain more insight on Android malware behavior. A large percentage of String manipulation operations are

performed on dangerous Linux commands (such as SU, mount, sh, bin, pm install, killall, chmod). These commands are mainly used by malware authors to root the phone and exploit some well known vulnerabilities. After getting superuser privilege, malwares perform various dangerous Linux operations through invoking runtime.exec(). Most of the ContentResolver operations are performed on SMS, MMS, telephony or call log content providers.

Class	Method	Parameter type	Difference (%)
StringBuilder	append	Dangerous command	35.95
ContentResolver	query	SMS or MMS	23.65
StringBuilder	append	Unicode string	23.6
StringBuilder	init	Dangerous command	23.07
DataOutputSream	writebytes	Reads from process	21.80
DataOutputSream	init	Reads from process	21.62
runTime	exec	Dangerous command	21.27
InetSocketAddress	init	Port 80	19.91
StringBuilder	append	Compressed file	19.58
DataInputStream	init	Reads from connection	19.27
String	valueOf	Unicode string	18.05
StringBuilder	append	File manupilation	17.79
File	init	Accesses external storage	16.92
InetSocketAddress	init	Explicit IP	14.87
String	getBytes	URL manupilation	14.05
Intent	setFlags	SendTo	12.94
Intent	setFlags	Call	11.67
ContentResolver	query	Telephony	10.88
Intent	setFlags	Send	10.47
ContentResolver	query	Call_log	10.12

Table A.2: Some Frequent API Parameters in Malware

A.4 Classification and Evaluation

A.4.1 Data Set

To extract malware and benign apps' features, generate and evaluate the classification models, I have collected and analyzed around 20,000 apps. Our malware sample consists of 3987 malware apps that I collected from different sources (McAfee and Android Malware Genome Project [139]). The malware sample belongs to different Android malware families. Our benign sample consists of the top 500 free apps in each category in Google Play (around 16000 apps) that I collected in July 2012.

A.4.2 Classification Models

As discussed earlier, our objective is to build a model that classifies unknown apps as either benign or malware. For that, I have employed four different algorithms for the classification: ID3 DT [131], C4.5 DT [131], KNN [133], and linear SVM [134]. These inducers belong to different family of classifiers. C4.5 and ID3 are related to decision trees and KNN belong to Lazy classifiers. SVM is a supervised learning method that proceeds through dividing the training data by an optimal separating hyperplane. I have decided to employ algorithms from different classifiers because I hope that they will produce different classification models for Android apps. Our analysis shows that KNN and ID3 DT models lead to a better accuracy compared to the other models.

To test our generated classification models, I use split validation. That is, I randomly split our dataset into training (2/3) and testing set (1/3). I build the classification models

based on the training set and feed the testing instances to evaluate the models. To evaluate each classifier's performance, I measured the True Positive Ratio (TPR), i.e., the proportion of malware instances that were correctly classified:

$$TPR = \frac{TP}{TP + FN}$$

where TP is the number of malware apps correctly identified and FN is the number of malware apps classified as benign apps. Similarly, I measure the True Negative Ratio (TNR), i.e., the proportion of benign instances that were correctly classified:

$$TNR = \frac{TN}{TN + FP}$$

where TN is the number of benign apps correctly identified and FP is the number of benign apps identified as malware apps. To capture the overall performa I nce, I measure the models' accuracy, i.e., the total number of benign and malware instances correctly classified divided by the total number of the dataset instances:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

By means of our collected dataset, I conducted different experiments to find the optimum feature set that will produce the best cut between the malware and benign sample.

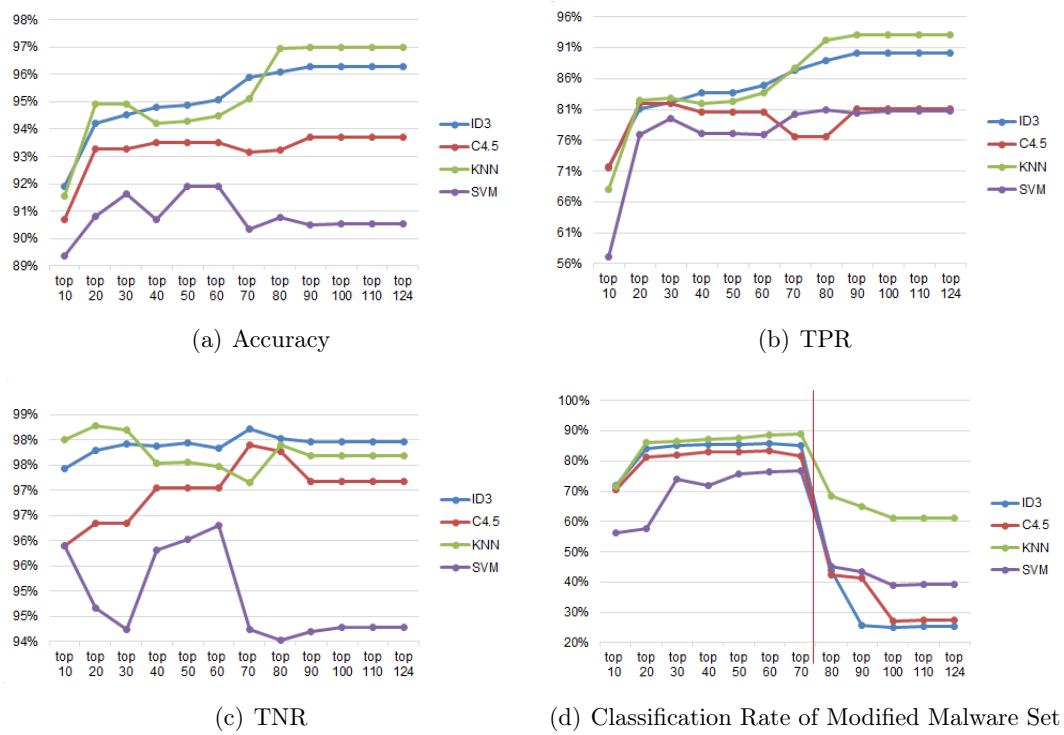
A.4.3 Permission-Based Feature Set

In the first experiment, I extract the permissions requested by malware and benign apps and obtain their perspective percentage usage in the two sets. I then rank the permissions based on the difference usage and took the top k permissions that are more frequently requested in malware than in benign apps. To determine the optimum k permissions, I evaluate the performance of the models for k = 10, 20, 30..., up to 124.

Fig. A.3 depicts the results obtained for the permission-based feature set in terms of accuracy, TPR, and TNR. As illustrated, the models' accuracy increases as the feature set includes more permissions. It should be noted that only 64 permissions were more frequent in the malware set than in the benign set, which means that after the top 64 permissions, the classifiers start to learn also from the permissions that are frequent in the benign set. This makes the classifiers not solid enough since they can fail to detect malicious apps in the following two scenarios. First, malware authors can easily defeat the permission-based classifiers through merely declaring "benign" permissions in the manifest file. Second, the classifiers will not be able to correctly classify repackaged android malware; which is based on legitimate apps but embeds extra payload to achieve a malicious goal. The manifests of the repackaged apps include both the original permissions of the benign app and the permissions needed for the malicious behavior and thus confuse the classifiers.

To demonstrate that the permission model is not robust enough, I designed an experiment in which I modify our malware set and feed it to the classifiers. In each malware manifest, I declare 10 new permissions (the top 10 in the benign set) and keep everything else unchanged. As shown in Fig. A.3(d), when the feature set contains the

permissions used in the benign set, the classifiers are not able to correctly classify the malware set. In fact, using the top 80 permissions, the classification rate of KNN drops to 67% and of ID3 to 43%.



(a) Accuracy

(b) TPR

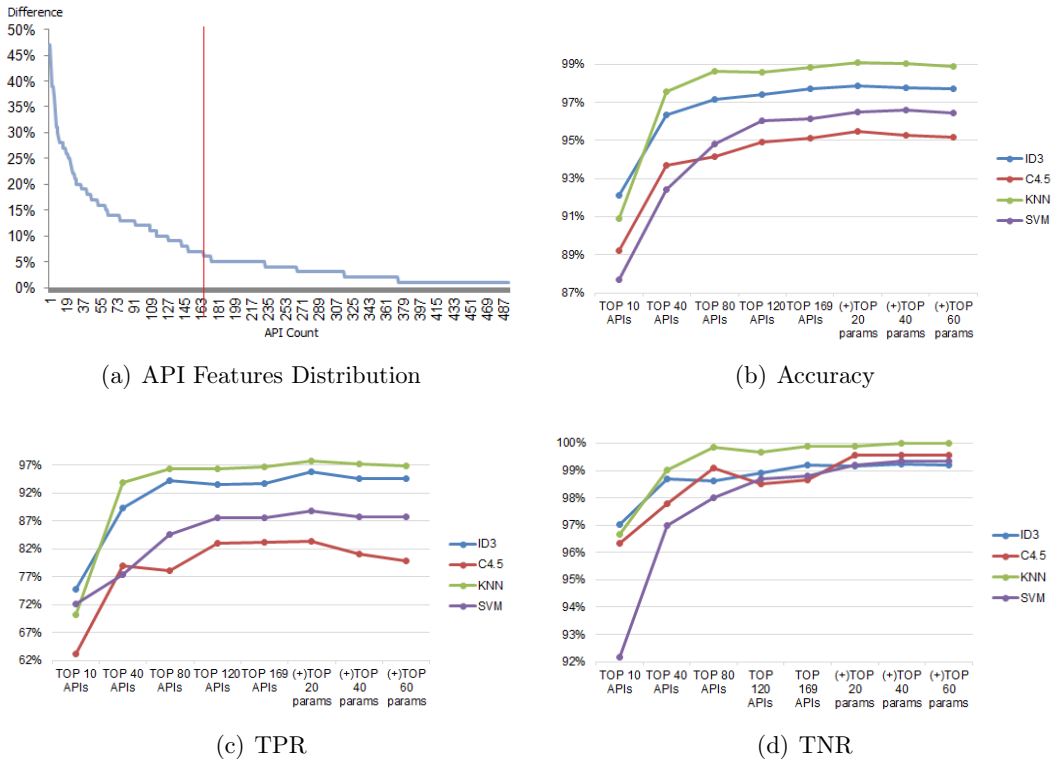
(c) TNR

(d) Classification Rate of Modified Malware Set

Fig. A.3.: Performance of Permission-based Models

A.4.4 API-Based Feature Set with Package Level and Parameter Information

In the second experiment, our feature vector includes the generated APIs within each set, which make up in total 8375 distinct APIs. I also embed package level information. That is, I white-list the APIs that are exclusively called by third-party packages. I specifically filter out these APIs to avoid the case where a benign app might be classified as



(a) API Features Distribution

(b) Accuracy

(c) TPR

(d) TNR

Fig. A.4.: Performance of API-based Models

malicious if a third-party package invokes a possibly "malicious"API. Consequently, the support of white-listed APIs drops in the benign set.

We conduct a frequency analysis and took only the APIs whose usage in the malware set is higher than in the benign set. Based on this, I have reduced our features to 491 APIs. As shown in Fig. A.4(a), a large portion of these APIs have a usage difference of less than 6% which will result in creating more noise in the classifiers and slow down the learning process. To solve this issue, I further refine our feature set to include only the top 169 APIs (with a usage difference greater or equal to 6%).

We generate the classification models for the top k (10, 40, 80, 120 and 169) API features and evaluate their performance. As depicted in Fig. A.4, using the top 169 API

based features, I achieve the highest accuracy, TPR and TNR using KNN. C4.5 is the worst performing model as it barely achieves 83% TPR.

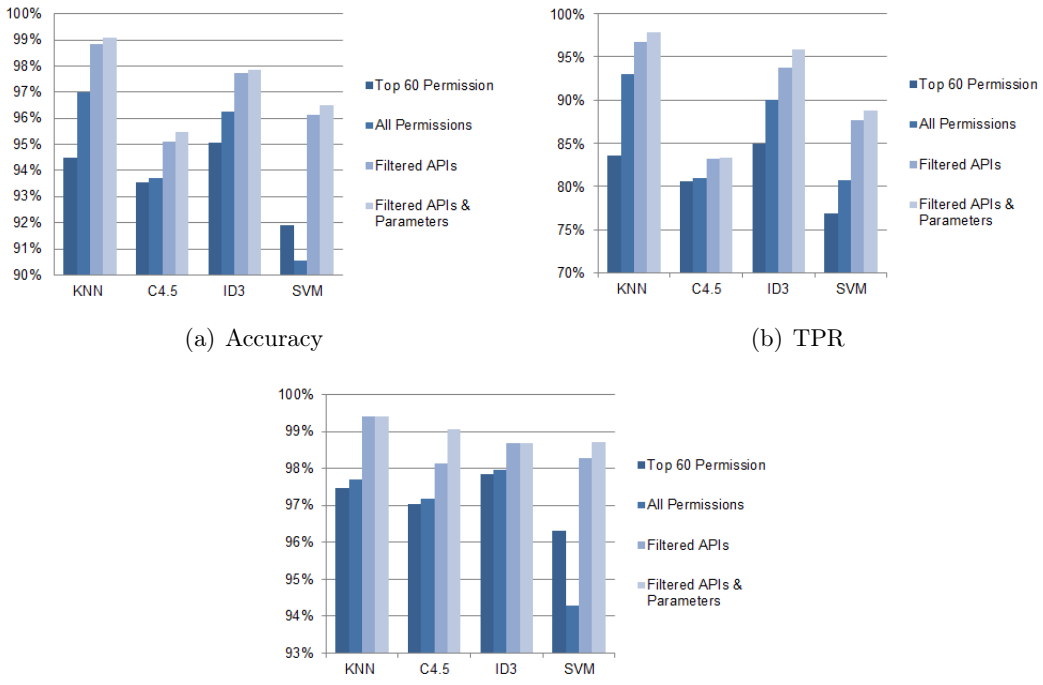
In the same experiment, I also include the parameter-based features obtained using data flow analysis on the original set. I re-generate the models and evaluate them after adding 20, 40, and 60 parameters to the 169 filtered APIs. As shown in Fig. A.4, by adding the top 20 used parameters, I are able to achieve the highest accuracy (99%) and TPR (97.8%) using KNN algorithm. The other algorithms also perform better with the newly added parameter-based feature set.

Unlike permission-based classifiers, it is not possible to trick API-based classifiers through declaring benign APIs, because the models do not rely on benign features to classify a given app. Rather, they only rely on the APIs (along with parameters) that are more frequently used in malware than in benign apps.

A.4.5 Models Comparison

To show the improvement achieved over the experiments performed, I plot the accuracy, TPR, and TNR of the classification models together as depicted in Fig. A.5. I consider two permission models. The first one is trained on the top 60 frequent permissions in malware and the second one on all the permissions. For the API filtered model, the feature vector includes all the top 169 features. The last model that I consider is trained on the top 169 filtered APIs along with the top 20 frequent parameters in certain APIs within malware.

As shown in Fig. A.5, our API based features performs better than the permission-based one. I were able to improve the accuracy, TPR and TNR of the models



(a) Accuracy

(b) TPR

(c) TNR

Fig. A.5.: Models Comparison

by embedding package and some parameter features to our original features. KNN is the best performing model, followed by ID3, SVM then C4.5.

A.4.6 Processing Time

It is evident that the processing time is a crucial metric for a scalable detection system. In this section, I report the execution time of DroidAPIMiner which consists of the time required to de-assemble an apk file and to extract the API and parameter feature set. I also report the time that RapidMiner requires for applying different classification models to classify a new instance. I perform the analysis an Intel Core i5-2430M machine with 6GB of memory.

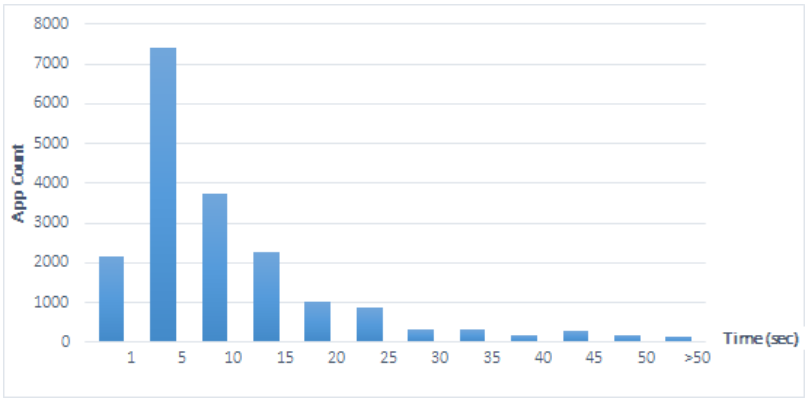


Fig. A.6.: Distribution of DroidAPIMiner Processing time

Fig. A.6 shows the distribution of DroidAPIMiner processing time among the collected apps sample. As depicted in the graph, more than 80% of the apps require less than 15 sec to be analyzed by DroidAPIMiner. Besides, as shown in Table. A.3 applying KNN algorithm to classify new inputs is quite fast and takes less than 10 sec. In total, our detection system requires on average about 25 sec to classify an apk file as either benign or malicious, which makes it efficient enough to be deployed on either mobile devices and back-end servers.

Algorithm	Model Application and Classification time (sec)
ID3	185.0 +- 32.0
KNN	9.0 +- 1.0
C4.5	21.0 +- 4.0
SVM	160.2 +- 40.0

Table A.3: Processing Overhead of the Classification Algorithms

A.5 Discussion

In this section, I discuss some potential evasion techniques that malware authors may adopt in order to thwart our classifiers. Furthermore, I discuss how our tool handles these cases.

- Reflection: Malware authors may use reflection to easily obfuscate any dangerous API call and thus evade the static detection of the occurrence of that API by our analysis tool. However, it should be noted that our study has shown that reflection APIs are more frequently used by our malware set than in the benign set, which makes them part of the feature vector for the classification.

- Native Code: To avoid static detectors at the bytecode level, malwares sometimes embed malicious payload within native content. Since our detection tool only works at bytecode level, it will not be able to detect any dangerous methods invoked. However, the use of JNI calls such as `System.loadLibrary()` is also used as a classification feature by our tool.

- Bytecode Encryption: To prevent reverse engineering of Java code, malware authors may encrypt their code and allow the decryption at runtime. Our tool considers decryption APIs as a classification feature.

- Dynamic Loading: As discussed earlier, DexClassLoader allows loading classes from .jar and .apk files at runtime and executing code not installed as part of an app. `loadClass()` in DexClassLoader also belongs to our feature set.

- More Benign Calls: Since our classifiers rely on the frequency of API calls, malware authors might think of introducing more benign API calls into their code. However, our tool is not susceptible to this problem, because I do not rely on the occurrence of benign API calls as a feature for the classification. Rather, I only consider the occurrence of malicious call as a feature.

A.6 Conclusion

I have presented a robust and lightweight approach for detecting Android malware based on different classifiers. To predict whether an app is benign or malicious, the classifiers rely on the semantic information within the bytecode of the applications ranging from critical API calls, package level information and some dangerous parameters invoked. Rather than following a heuristic based approach for determining the feature vector of the classifiers, I have statically analyzed a large corpus of Android malwares belonging to different families and a large benign set belonging to different categories. I have conducted a frequency analysis to capture the most relevant API calls that malware invoke, and refined the feature set to exclude API calls made by third-party packages. I performed a simple data flow analysis to get dangerous input to some API calls.

My classification results indicate that I am able to achieve a better accuracy, TPR and TNR using a combination of API, package, and parameter level information in comparison to the permissions-based feature set. As future work, I plan to further reduce the false positives and negatives through analyzing the samples that were not correctly classified and finding out the reasons behind the misclassification.

LIST OF REFERENCES

[1] "Smartphone OS Market Share, 2015 Q2." http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] "Android Fragmentation Visualized." http://opensignal.com/reports/2015/08/android-fragmentation/.

[3] M. C. 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.

[4] 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.

[5] 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*.

[6] 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.

[7] "Compatibility Program Overview." https://source.android.com/compatibility/overview.html.

[8] Samsung, "Samsung knox." https://www.samsungknox.com/en.

[9] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, (Washington, DC, USA), pp. 83–92, IEEE Computer Society, 2012.

[10] "Firebase test lab for android." https://firebase.google.com/docs/test-lab/?hl=en.

[11] D. R. Thomas, A. R. 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.

[12] "exploid udev." http://androidvulnerabilities.org/vulnerabilities/exploid_udev.

[13] "Gingerbreak." http://androidvulnerabilities.org/vulnerabilities/Gingerbreak.

[14] "Apk duplicate file." http://androidvulnerabilities.org/vulnerabilities/APK_duplicate_file.

[15] "Apk unchecked name." http://androidvulnerabilities.org/vulnerabilities/APK_unchecked_name.

[16] "Google announces new update policy for nexus devices including monthly security patches for 3 years and major otas for 2 years from release." http://www.androidpolice.com/2015/08/05/ google-announces-new-update-policy-for-nexus-devices-including-monthly-security-pa

[17] "Samsung announces an android security update process to ensure timely protection from security vulnerabilities." https://news.samsung.com/global/ samsung-announces-an-android-security-update-process-to-ensure-timely-protection-f

[18] "Compatibility definition." http://static.googleusercontent.com/media/source.android.com/en//compatibility/android-cdd.pdf, October 2015.

[19] "ANRED: Android Residue Detection Framework." https://goo.gl/Q0d5qH.

[20] A. P. Felt, H. J. 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.

[21] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, pp. 36–38, Oct. 1988.

[22] 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.

[23] R. Gallo, P. Hongo, R. Dahab, L. C. 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.

[24] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, ACM, 2012.

[25] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[26] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security Privacy, IEEE*, vol. 7, pp. 50–57, Jan 2009.

[27] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security symposium*, 2011.

[28] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: A survey of current android attacks," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2011.

[29] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *37th IEEE Symposium on Security and Privacy (S&P '16)*, IEEE, 2016.

[30] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 627–638, ACM, 2011.

[31] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 217–228, ACM, 2012.

[32] T. Vidas, N. Christin, and L. F. Cranor, "Curbing android permission creep," in *W2SP*, May 2011.

[33] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner, "How to ask for permission," in *Proceedings of the 7th USENIX Conference on Hot Topics in Security*, HotSec'12, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2012.

[34] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[35] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, (Berkeley, CA, USA), pp. 499–514, USENIX Association, 2015.

[36] B. Liu, J. Lin, and N. Sadeh, "Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help?," in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, (New York, NY, USA), pp. 201–212, ACM, 2014.

[37] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 527–542, USENIX Association, 2013.

[38] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 1354–1365, ACM, 2014.

[39] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for android apps," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 518–529, ACM, 2015.

[40] 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.

[41] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 1017–1028, ACM, 2013.

[42] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *20th USENIX Security Symposium*, (San Francisco, CA), Aug. 2011.

[43] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, (New York, NY, USA), pp. 49–54, ACM, 2011.

[44] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11, (Berlin, Heidelberg), pp. 93–107, Springer-Verlag, 2011.

[45] 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.

[46] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[47] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, 2009.

[48] 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.

[49] 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.

[50] 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, D.C.), pp. 131–146, USENIX, 2013.

[51] 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.

[52] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. S. 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.

[53] 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 Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pp. 1033–1040, 2011.

[54] 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.

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

[56] S. Shekhar, M. Dietz, and D. S. 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.

[57] X. Zhang, A. Ahlawat, and W. Du, "AFrame: Isolating Advertisements from Mobile Applications in Android," in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, (New Orleans, Louisiana, USA), December 9-13 2013.

[58] 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.

[59] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," ACSAC '11.

[60] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 635–646, ACM, 2013.

[61] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "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*, CCS '14, (New York, NY, USA), pp. 66–77, ACM, 2014.

[62] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, (New York, NY, USA), pp. 239–252, ACM, 2011.

[63] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, "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*, CCS '14, (New York, NY, USA), pp. 978–989, ACM, 2014.

[64] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du, "Life after app uninstallation: Are the data still alive? data residue attacks on android," in *NDSS*, 2016.

[65] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, "Hey, you, get off of my clipboard," in *In proceeding of 17th International Conference on Financial Cryptography and Data Security*, 2013.

[66] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, (New York, NY, USA), pp. 73–84, ACM, 2013.

[67] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, (New York, NY, USA), pp. 659–668, ACM, 2013.

[68] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1236–1247, ACM, 2015.

[69] H. Zhang, D. She, and Z. Qian, "Android root and its providers: A double-edged sword," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1093–1104, ACM, 2015.

[70] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 97–112, USENIX Association, 2013.

[71] C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets," in *21st Annual Network and Distributed System Security Symposium (NDSS)*, The Internet Society, 2014.

[72] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: Ui state inference and novel android attacks," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 1037–1052, USENIX Association, 2014.

[73] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: Your finger taps have fingerprints," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, (New York, NY, USA), pp. 323–336, ACM, 2012.

[74] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, (New York, NY, USA), pp. 113–124, ACM, 2012.

[75] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in android," in *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, (Berkeley, CA, USA), pp. 945–959, USENIX Association, 2015.

[76] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp)iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 551–562, ACM, 2011.

[77] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion," in *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, HotSec'11, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2011.

[78] A. Al-Haiqi, M. Ismail, and R. Nordin, "On the best sensor for keystrokes inference attack on android," in *The 4th International Conference on Electrical Engineering and Informatics (ICEEI)*, Procedia Technology, 2013.

[79] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 1053–1067, USENIX Association, 2014.

[80] M. Azizyan, I. Constandache, and R. Roy Choudhury, "Surroundsense: Mobile phone localization via ambience fingerprinting," in *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, (New York, NY, USA), pp. 261–272, ACM, 2009.

[81] Y. Michalevsky, G. Nakibly, A. Schulman, and D. Boneh, "Powerspy: Location tracking using mobile device power analysis," in *24th USENIX Security Symposium*, 2015.

[82] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 235–245, ACM, 2009.

[83] B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android Permissions: A Perspective Combining Risks and Benefits," *SACMAT*, 2012.

[84] 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.

[85] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android Application Sandbox System for Suspicious Software Detection," *MALWARE*, 2010.

[86] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," *NDSS*, 2012.

[87] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J.Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static Analysis of Executables for Collaborative Malware Detection on Android," *ICC*, 2009.

[88] I. Burguera, U. Zurutuza, and S.Nadijm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android.," *SPSM*, 2011.

[89] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by 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.

[90] R. Potharaju, A. Newell, C. Nita-Rotaru, , and X. Zhang, "Plagiarizing Smartphone Applications: Attack Strategies and Defense," *ESSoS*, 2012.

[91] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, (New York, NY, USA), pp. 317–326, ACM, 2012.

[92] 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.

[93] 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.

[94] 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.

[95] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, pp. 5:1–5:29, June 2014.

[96] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile Protection for Smartphones," *ACSAC*, 2010.

[97] 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.

[98] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in droidsafe," 2015.

[99] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," in *NDSS*, 2014.

[100] M. Zhang and H. Yin, "Efficient, 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.

[101] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs.," in *NDSS*, The Internet Society, 2013.

[102] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: 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.

[103] 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.

[104] L. K. 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.

[105] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "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.

[106] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors.," in *NDSS*, The Internet Society, 2015.

[107] M. Lindorfer, M. Neugschw, L. Weichselbaum, Y. Fratantonio, V. V. D. 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.

[108] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *CCS*, (New York, NY, USA), ACM, 2013.

[109] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, Nov. 2012.

[110] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *NDSS*, 2016.

[111] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *NDSS*, 2016.

[112] P. Brodley and leviathan Security Group, "Zero Permission Android Applications." `https://www.leviathansecurity.com/blog/zero-permission-android-applications/`. Accessed: 10/02/2013.

[113] P. Ratazzi, Y. Aafer, A. Ahlawat, H. Hao, Y. Wang, and W. Du, "A systematic security evaluation of Android's multi-user framework," in *Mobile Security Technologies (MoST) 2014*, MoST'14, (San Jose, CA, USA), May 17 2014.

[114] "Soot." `http://sable.github.io/soot/`.

[115] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis," *arXiv preprint arXiv:1404.7431*, 2014.

[116] "Samsung Updates." `http://goo.gl/RVU84V`.

[117] "Android revolution mobile device technologies." `http://android-revolution-hd.blogspot.com/p/android-revolution-hd-mirror-site-var.html`. Last Accessed: May 13, 2015.

[118] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "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*, CCS '15, (New York, NY, USA), 2015.

[119] "Huawei ROMs." http://goo.gl/dYPTE5.

[120] "Android Revolution." http://goo.gl/MVigfq.

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

[122] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," *CCS*, 2009.

[123] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, and R. Potharaju, "Using Probabilistic Generative Models for Ranking Risks of Android Apps," *CCS*, 2012.

[124] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission Evolution in the Android Ecosystem," *ACSAC*, 2012.

[125] A. P. Felt, K. Greenwood, and D. Wagner, "The Effectiveness of Application Permissions," *USENIX, WebApps*, 2011.

[126] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," *CCS*, 2011.

[127] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," *NDSS*, 2012.

[128] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," *CCS*, 2012.

[129] "Androguard," http://code.google.com/p/androguard/.

[130] "RapidMiner," http://rapid-i.com/content/view/181/190/.

[131] J. R. Quinlan, "Induction of Decision Tree," *Machine Learning, Vol. 1, No. 1. pp. 81-106.*, 1986.

[132] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[133] D. W. Aha, D. Kibler, and M. K. Albert, "Instance-Based Learning Algorithms," *Machine Learning, 6:37–66*, 1991.

[134] V. Vapnik., *The Nature of Statistical Learning Theory*. Springer-Verlag, NY, 1995.

[135] "Malware that Takes Without Asking," http://labs.mwrinfosecurity.com/tools/2012/03/16/mercury/documentation/white-paper/malware-that-takes-without-asking/.

[136] "Intent," http://developer.android.com/reference/android/content/Intent.html.

[137] "ActivityManager," http://developer.android.com/reference/android/app/ActivityManager.html.

[138] "Process," http://developer.android.com/reference/android/os/Process.html.

[139] "Android Malware Genome Project," http://www.malgenomeproject.org/.

VITA

Yousra Aafer

Syracuse University Graduate Department of Computer Science

yaafer@syr.edu

111 Lafayette Road, Syracuse, NY, 13205

Education

- **Ph.D. in Electrical and Computer Engineering**
  Syracuse University, USA, August 2016
- **M.S. of Science in Computer Engineering**
  Syracuse University, USA, December 2011
- **B.S. in Computer Science**
  Al Akhawayn University, Morocco, December 2009

.

EXPERIENCE

- **Syracuse University.**, 01/15 - 07/16, **Research Assistant:** Conducted research on Android customization hazards and hanging attribute (Hare) references vulnerability.
- **SEED Workshop.**, 06/2015 and 06/2016, **Organizer:** Helped organize SEED workshop at Syracuse University during June'15 and June'16.
- **Samsung Research America, Knox Team:**, 09/14 - 11/14, **Research Intern:** Worked on analyzing the Knox security container from several security aspects.
- **Microsoft, Inc.**, 05/14 - 08/14, **Software Engineer Intern:** Developed a debugger extension that allows de-obfuscating Windows binaries to help Microsoft internal developers better understand reported software crashes.

- **Syracuse University.**, 01/12 - 05/14, **Teaching Assistant:** Help instructors organize classes and lead lab sessions, including both graduate level courses (Algorithms, Computer Architecture and Data Mining) and undergraduate level courses (Data Structures and Algorithms).

AWARDS

- **Practical Application Winner**: NUNAN Research Competition, EECS Dept., Syracuse University, April 6, 2015.
- **Department Winner**: NUNAN Research Competition, EECS Dept., Syracuse University, April 6, 2015.
- **Best Capstone**: Computer Science Dept., Al Alakhawayn University, June, 2010.

PUBLICATIONS

---

1. **Hey, You, Get Off of My Image: Detecting Data Residue in Android Images**,
   X. Zhang, Y. Aafer, K. Ying and W. Du,
   To appear in Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS'16). Heraklion, Crete, Greece. September 26-30, 2016.
2. **Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis**,
   Y. Aafer, X. Zhang, and W. Du,
   To appear in Proceedings of the 25th USENIX Security Symposium (USENIX Security'16), Austin, Texas, USA. August 10-12, 2016.
3. **Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android**,
   X. Zhang, K. Ying, Y. Aafer, Z. Qiu and W. Du,
   In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, California, USA. February 21-24, 2016.
4. **Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References**,
   [Y. Aafer, N. Zhang]co-first author, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace,
   In Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS), Denver, Colorado, USA. October 12-16, 2015.
5. **A Systematic Security Evaluation of Android's Multi-User Framework**,
   P. Ratazzi, Y. Aafer, A. Ahlawat, H. Hao, Y. Wang and W. Du,
   In Proceedings of the Mobile Security Technologies (MoST) workshop, May 16, 2014.
6. **DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android**,
   Y. Aafer, W. Du and H. Yin
   In Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm). September 25-27, 2013 Sydney, Australia.