December 2016

Understanding and Improving Security of the Android Operating System

Edward Paul Ratazzi

Syracuse University

ABSTRACT

Successful realization of practical computer security improvements requires an understanding and insight into the system's security architecture, combined with a consideration of end-users' needs as well as the system's design tenets. In the case of Android, a system with an open, modular architecture that emphasizes usability and performance, acquiring this knowledge and insight can be particularly challenging for several reasons. In spite of Android's open source philosophy, the system is extremely large and complex, documentation and reference materials are scarce, and the code base is rapidly evolving with new features and fixes. To make matters worse, the vast majority of Android devices in use do not run the open source code, but rather proprietary versions that have been heavily customized by vendors for product differentiation.

Proposing security improvements or making customizations without sufficient insight into the system typically leads to less-practical, less-efficient, or even vulnerable results. Point solutions to specific problems risk leaving other similar problems in the distributed security architecture unsolved. Far-reaching general-purpose approaches may further complicate an already complex system, and force end-users to endure significant performance and usability degradations regardless of their specific security and privacy needs. In the case of vendor customization, uninformed changes can introduce access control inconsistencies and new vulnerabilities. Hence, the lack of methodologies and resources available for gaining insight about Android security is hindering the development of practical security solutions, sound vendor customizations, and end-user awareness of the proprietary devices they are using.

Addressing this deficiency is the subject of this dissertation. New approaches for analyzing, evaluating and understanding Android access controls are introduced and used to create an interactive database for use by security researchers as well as system designers and end-user product evaluators. Case studies using the new techniques are described, with results uncovering problems in Android's multiuser framework and vendor-customized System Services. Finally, the new insights are used to develop and implement a novel virtualization-based security architecture that protects sensitive resources while preserving Android's open architecture and expected levels of performance and usability.

UNDERSTANDING AND IMPROVING SECURITY OF THE ANDROID OPERATING SYSTEM

by

Edward Paul Ratazzi

B.S., Rensselaer Polytechnic Institute, 1987

M.S., Syracuse University, 1992

M.S., Rensselaer Polytechnic Institute, 2006

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical & Computer Engineering

Syracuse University

December 2016

DISCLAIMER

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

Dedicato a mio nonno, Edward Ratazzi, Sr.

—

Dedicated to my grandpop, Henry Paul, Jr.

Acknowledgments

My deepest gratitude goes to those around me who made completing this dissertation possible.

First, to my advisor, Prof. Wenliang Du. Even though I met you with a career's worth of experience already behind me, your insights about conducting research, distilling problems and critical thinking have changed my professional life. I will consider myself a great success if I can pass along to others even a fraction of what I learned from you. I am especially thankful for your patience, approachability, friendly style, and understanding of my outside commitments to work and family.

To my defense committee, Prof. Joon Park, Prof. Shiu-Kai Chin, Prof. Jian Tang, Prof. Yuzhe Tang, and Prof. Heng Yin, for taking time out of your busy schedules to read this dissertation, provide valuable feedback and serve on my committee.

To the Information Directorate of the Air Force Research Laboratory for its commitment to career-long learning and professional development. To my past and present colleagues there, including Dr. Warren Debany, Jr., Dr. Kamal Jabbour, Dr. Davy Belk, Joe Camera, Lt. Col. David Bibighaus, Dr. Dan Pease, and Dr. Lok Yan. You supported and guided my return to graduate school, and provided much-needed encouragement along the way. I am particularly indebted to my supervisor and friend, Jim Perretta. For the last four years you have sheltered me from many day-to-day distractions so that I could focus on conducting and documenting my in-house research. Now that I am done, I hope to rise to the new challenges and opportunities your leadership brings to me.

To the current and former students of the Computer Security Research Group at Syracuse, especially Amit Ahlawat, Francis Akowuah, Ashok Bommisetti, Nian Ji, Dr. Yousra Aafer, Dr. Xiao Zhang, Jiaming Liu, Kailiang Ying, Yifei Wang, Hao Hao, Haichao Zhang, and Lusha Wang. I am in awe of your technical skills and grateful for the countless hours of discussions we've had, both in group meetings and one-on-one. I wish you all the best and hope we can collaborate again in the future.

To everyone at the Griffiss Institute, especially Bill Wolf, Regan Johnson, Tracy DiMeo, Dr. Josh White, and Jim Hanna. You provided a comfortable, quiet and well-connected environment in which to study, research, collaborate, and write. Without your support, completing this dissertation would have been tremendously more difficult and lengthy.

To my parents, Randa and Ed Ratazzi. By example, you taught me the value of education, the need for perseverance, and the importance of optimism, all ingredients I found to be essential for completing my studies.

Finally and most importantly, to my wife Shirley and children Emily and Nicholas. You encouraged me when things were tough, cheered my successes, and made countless sacrifices along the way. Your love and confidence are the cornerstone of this and all other accomplishments of mine.

Syracuse, New York

December 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I for one welcome our new android overlords.

- Robin Sloan, *Mr. Penumbra's 24-Hour Bookstore*

Over the near-decade since its 2008 introduction, Google's Android has been a stunning success, eclipsing the market share of every other mobile operating system by a huge margin. Android is shipping on well over *1 billion* new devices annually [1], and over 1.5 million new devices are being activated *every day* [2]. This growth, however, has not been without its pains. Recent measures estimate that 96-97% of today's mobile malware targets the Android operating system [3,4], and 73% of these are designed specifically to satisfy profit motives [4]. Also, as the system becomes more popular and scrutinized, the number of identified vulnerabilities has skyrocketed. For the years 2009-2014, the National Vulnerability Database (NVD) maintained by the National Institute of Technology (NIST) issued a total of 43 Common Vulnerabilities & Exposures (CVE) reports for Android. In 2015, this jumped to 125, and 2016 had seen 346 by August with four months still to go [5].

In light of this, end-users are increasingly concerned about privacy and protecting their personal information. Unfortunately, most have a hard time using available security indicators to discern the trustworthiness of apps [6]. Even if they can determine that a particular app warrants extra caution, many users possesses little or none of the specialized technical expertise necessary to fully understand the relevant security controls. Finally, secure modes of operation often degrade performance or usability, sometimes to levels that far outweigh users' willingness to compromise. Faced with this lack of information, complexity, and unattractive trade-offs, many users become complacent, careless, or make mistakes in performing what amounts to critical system administration tasks.

Security researchers and developers are also working hard to address the problems with Android. Although successive releases continue to enhance and improve security and user controls [7], balancing security with usability has proven difficult for system designers and security researchers. For example, while evidence of the shortcomings of Android's permission system has been present in the literature as early as its initial release [8], no real user control over permission granting was provided until the advent of the ill-fated *App Ops* hidden feature in July 2013. This selective permission-granting mechanism was removed less than 6 months later, apparently due to usability concerns that quickly surfaced when it was first released [9]. Because developers could not anticipate the endless security configurations *App Ops* makes possible, many apps failed to function or simply crashed when their permissions were selectively revoked by the user.

Although *App Ops* was technically sound, and many useful 3rd party apps were created to "unhide" it, Google removed it nonetheless. To understand why a useful security feature was removed (much to the chagrin of security and privacy advocates [9]), the original goals of the Android project must be revisited. Android's design is based on several strong tenets that are unlikely to be put aside, even for dramatically increased security or privacy. According to AOSP's on-line security documentation [10]:

- "Android is a modern mobile platform that was designed to be truly open."
- "Android was designed with developers in mind. Security controls were designed to reduce the burden on developers."

- “Android was designed with device users in mind. Users are provided visibility into how applications work, and control over those applications.”
- “Android seeks to be the most secure *and usable* operating system for mobile platforms...” (emphasis added).

In the case of *App Ops*, Google’s commitment to making things easy for developers and end-users outweighed security, and it was removed. After all, as Felten and McGraw state in [11], “given a choice between dancing pigs and security, users will pick dancing pigs every time.” Because of Google’s caution in this regard, even later releases of Android 5.0 did little to help end-users protect themselves from software they chose to install. Only with the advent of Android 6.0 did selective permissions and run-time confirmations become part of the Android mainstream [12,13]. With this example, we can see that developing and deploying viable security and privacy improvements in Android requires considerations beyond those that are purely technical. Taking a look at other examples of Android security enhancements, we consider two categories: those proposed via the scientific literature, and those present in actual products and apps available to end-users.

1.1 Security enhancements proposed by the scientific literature

Going beyond the *App Ops* example, it’s not hard to conclude that Android’s architects are unlikely to embrace other solutions to security problems that would make the system more closed or restrictive, increase developers’ burden, relinquish control from or inconvenience users (to include noticeable performance degradation). If technical improvements are pursued without consideration of these tenets, the result may run counter to one or more of them, and most likely prevent its widespread adoption. In fact, there’s no shortage of interesting and sound technical Android security improvements in the scientific literature that have had no direct impact on the open source project, which instead has improved incrementally and with bug fixes for specific vulnerabilities [7]. Obviously the many proposed solutions found in the literature have technical merit or they wouldn’t have appeared in peer-reviewed forums. However, from Google’s point of view most are impractical because they represent radical system redesigns or run counter to the vision of Android being open and easy to use and develop for. In addition, many fail to account for end-users’ specific security and privacy needs and instead trade usability, performance and even basic device functionality for security that the user may not want or need. A more detailed discussion of these related works is contained in Chapter 6.

1.2 Security enhancements available to end-users

Although AOSP may shun security improvements that have negative effects on usability or performance, even if they are technically sound, there are a number of products that are successfully marketed to end-users via Google Play as “add ons”. These are successful because they “fit in” with Android’s design tenets, and are perceived by end-users as filling an important security need. One example is Android virus detectors (AVD), the most popular of which have as many as 500 million downloads from Google Play [14].

AVDs are examples of security solutions that naively apply traditional (i.e., PC) security concepts to Android, without adapting to Android’s fundamental differences. A great deal of their success in the marketplace is due to end-users’ fear coupled with a misunderstanding of their theory of operation. Specifically, the very basis for anti-virus’ effectiveness on PCs, 3rd party access to administrative privileges, doesn’t exist on production Android devices [15]. Thus, AVDs are limited to providing warnings based on the limited system information Android makes available to apps. Moreover, the current design of Android’s *ActivityManager*, *PackageManager*, system broadcasts, and various Linux facilities such as **`build.prop`** and **`/proc`** filesystem, can leak critical information about an AVD’s running state to malware, dramatically increasing the chance of AVD evasion [16]. Google has addressed this somewhat with its centralized malware scanning strategy for the Play Store, but users are able to negate this by allowing *Unknown sources* or disabling *Verify apps*. Chapter 2 includes a discussion of other important differences that must be addressed when developing security products for mobile devices.

Whether it be sound-yet-impractical proposals from the research community, or easily-installable security apps that give a false sense of security, it is clear that developing practical and effective security for Android requires three things:

1. Adherence to Google's design tenets which emphasize openness and usability.
2. Consideration of users' and developers' interests in terms of security needs and the burden imposed.
3. Thorough technical knowledge of the system, especially that related to its unique architecture, characteristics, and existing security mechanisms.

Although the design tenets are relatively straightforward, and users' security needs are readily identified, acquisition of the unique system knowledge necessary for security insights is difficult and and time-consuming for three reasons.

First, the Android system is extremely complex. In April 2016, the Black Duck Open Hub free and open source software (FOSS) directory reported that the Android Open Source Project (AOSP) includes 13,499,670 lines of code (LoC) representing 4,271 years of effort by 3,448 contributors. This massive code base is composed of 36 different programming languages, with the vast majority written in Java (40%), C (29%), and C++ (18%) [17]. Besides making the system difficult to understand, this complexity also has ramifications to security. As Bruce Schneier remarked in 2012, "complexity is the worst enemy of security" [18]. When this complexity is coupled with a distribution and stakeholder model that makes rapid patching nearly impossible, the real-world impact of the problem is amplified. Indeed, a 2015 study revealed that overall, at any given moment, a majority of Android devices are running a vulnerable version [19].

Second, access controls in Android use multiple means of identifying the subject, are implemented across multiple layers of software, and usually reside with the object being accessed rather than in a centralized reference monitor. This makes identification of all factors in an access control decision difficult. For example, for an app to access location, it must first obtain permission from the user. This is accomplished via a dialog box that identifies the app to the user making the decision by common name. Next, a binder token issued by the kernel binder driver after consideration of the requester's *uid* and *pid* allows the app to communicate with the native *ServiceManager* running in userspace and obtain the location service handle. Distribution of some of these service handles are restricted to specific `uid`s and is enforced by the native code. Next, when the app uses the handle to request location from the service, an `IPC_DESCRIPTOR` value from each side of the IPC channel is compared to ensure the calling process's transaction matches the interface of the reciever. Next the caller's manifest permissions are checked by the specific service thread running in the native `systemserver`, again using the `uid` and `pid`. Finally, a "location blacklist" is consulted using the package name. Underlying each of these steps is enforcement of mandatory access controls (MAC) determined by the device's SE for Android (i.e., SELinux) policy. With so many checks and decisions spread out across many parts of the system, addressing security or privacy concern with apps' use of location is difficult and the solutions can take many forms with various tradeoffs.

Lastly, security documentation of the Android system leaves much to be desired, which results in more mistakes by developers and end-users, while increasing the learning challenges for security researchers. Although the situation has improved since, a 2011 study found that only 6.2% (78/1,259) of application programmer interface (API) calls with permission checks are documented, and of those, nearly 8% (6/78) were incorrectly documented [20]. Moreover, official documentation of Android internals is virtually non-existent, making understanding how access controls and security mechanisms are implemented within the vast code base a daunting task. Doing so requires painstaking source code review and consultation with informal (and possibly incorrect or outdated) sources such as blogs, forums and on-line tutorials. Finally, for vendor customizations of Android, gaining this knowledge is even more difficult since vendor code and their modifications are typically closed-source. In fact, the vast majority of devices in use are not running pure Android, but many different versions each customized differently by numerous competing vendors.

1.3 Thesis and Contributions

Providing the means to acquire the necessary system knowledge and insight so that practical security and privacy improvements can be designed for Android is the subject of this dissertation. This dissertation's thesis is as follows:

> *It is possible to design and implement practical security and privacy improvements for Android when the following conditions are met:*
>
> 1. *Insights are gained from a systematic understanding of Android security and its unique characteristics;*
> 2. *End-users' security and privacy needs are accounted for; and*
> 3. *Android's design tenets are preserved.*

Because of its complexity, Android system understanding requires a *systematic* approach to gathering knowledge. Chapter 3 describes the development of such an approach and its application to the first and only known security evaluation of Android's multi-user architecture.

In spite of the successes derived from this effort, it's clear that manual analysis of the system is too time-consuming and error-prone to be practical for a platform like Android that evolves so rapidly. The problem is further compounded when vendors make proprietary modifications to the platform in order to customize and differentiate their product. The lack of source code in these cases makes gathering the required knowledge difficult at best. Addressing this problem led to the development of an efficient, automated modeling and analysis technique described in Chapter 4. This technique enables security researchers to gain insight about the design of access controls in System Services, rapidly identify specific vendor customizations of System Services, and focus their testing on the most interesting portions of code.

Combined, these efforts yielded deep insights regarding the protection, or lack thereof, of common system resources in Android and its vendor-specific derivatives. Unfortunately, directly addressing many of the problematic situations uncovered would be difficult due to Android's open design and the number of stakeholders involved. For example, while patching a vendor's insecure System Service with an appropriate access control might fix that specific problem, it would do nothing to address future modifications by other vendors or stakeholders that may make mistakes. Instead, a more general solution was sought. Chapter 5 describes *PINPOINT*, a novel, lightweight isolation-based solution to protect sensitive, or vulnerable, resources and information from untrusted apps. The effectiveness of PINPOINT is independent of vendor customizations and any shortcomings of resources' access controls. Moreover, PINPOINT offers a practical alternative to existing isolation architectures that force users to sacrifice convenience, performance and usability for security features they may not need.

In all, this dissertation describes the following contributions:

1. **An insightful model and systematic methodology for analyzing access controls in Android** (see Chapter 3, Section 3.3).
2. **The first and only known systematic security evaluation of Android's multiuser architecture**, also serving as a case study to demonstrate the efficacy of the aforementioned methodology (see Chapter 3, Section 3.4).
3. **A novel Android access control feature set**, useful for characterizing Android access controls (see Chapter 4, Section 4.2).
4. A feature set-based **differential analysis methodology**, useful for identifying and understanding closed-source vendor modifications to Android System Services (see Chapter 4, Section 4.3).
5. **A case study of 12 closed-source vendor images**, that applies the feature-set based differential analysis to reveal specific shortcomings and actual access control issues

stemming from vendor modifications to Android System Services (see Chapter 4, Section 4.4).

6. **Two access control feature databases** containing over 35,000 methods from System Services of 19 Android images, allowing security researchers to interactively query and compare System Service access controls.
7. **A novel, lightweight *hypovisor*-based approach to protecting sensitive and vulnerable resources from untrusted apps and sensitive apps from untrusted resources.** Known as PINPOINT, the concept deliberately addresses Android's unique open architecture, while simultaneously respecting its design tenets (see Chapter 5, Section 5.3).
8. **A working prototype implementation of PINPOINT** that demonstrates a new isolation capability for the full range of Android System Services. The prototype is evaluated using four System Service case studies (see Chapter 5, Sections 5.4 and 5.5).

1.4 Dissertation organization

The remainder of this dissertation is organized as follows:

- Chapter 2 includes background material on Android security and discusses some important characteristics of the Android architecture;
- Chapter 3 describes the systematic access control analysis methodology as well as the multiuser case study;
- Chapter 4 describes the design of the access control feature set, feature extraction methodology, and feature vector-based case study of 19 real Android images;
- Chapter 5 describes PINPOINT, its design, implementation, application and evaluation;
- Chapter 6 discusses previous work by others and relates it to the contributions claimed; and
- Chapter 7 concludes the dissertation and proposes some future opportunities.

Chapter 2

Background

> *I wonder, he wondered, if any human has ever felt this way before about an android.*
>
> - Philip K. Dick, *Do Androids Dream of Electric Sheep?*

Since this dissertation focuses on understanding and improving security of Android devices, this chapter gives a brief overview of the unique characteristics impacting mobile security architectures, followed by a tutorial on Android security. Background information specific to material discussed only in a single chapter is included within the respective chapter.

2.1 Uniqueness of Mobile Devices

Android is an operating system designed specifically for lightweight mobile devices. As such, it has unique characteristics that set it apart from traditional platforms. Identifying and understanding these differences is an important prerequisite when studying the system and proposing improvements.

Table 2.1 compares mobile devices with fixed computers in terms of features, characteristics, and environment. Each of these has an impact on security and potentially the design of the platform's

Table 2.1: A comparison of mobile vs. fixed computing platform attributes.

	Typical attributes	
Category	**Mobile**	**Fixed**
Input	Soft keyboard, tap	Full keyboard, mouse, tap
Authentication	PIN, pattern, short password, fingerprint	Long password, multi-factor
Sensors	Motion, position, sound, light, etc.	None
Media	Multiple high-resolution cameras	Webcam
Communication	4G, cellular, Bluetooth, NFC, WiFi, peer-to-peer, etc.	Ethernet
Connectivity	Traverse multiple untrusted/insecure networks	Direct, trusted connection
Enterprise connectivity	VPN or direct-app connection to enterprise	Direct, trusted connection
Administration	Lacks “root” access; OS limits security apps	Full administrative access
Patch/update	Carrier- and device-dependent	Owner-dependent
Special uses	ID/financial proxy, billed services (e.g., SMS)	
Physical security	Vulnerable to loss, theft	Secured

security architecture.

The lack of a full-size keyboard on mobile devices makes entering long, complex passwords difficult as well as inconvenient. As a result, several other authentication options are typical of mobile devices. However, not all of these are as secure as the strong passwords and multi-factor authentication common to fixed systems. Moreover, the combination of these with sensors and other features unique to mobile devices can further exacerbate security problems. For example, it has been shown that lock screen PINs can be inferred through a device’s camera and microphone [21] or by other sensors such as the accelerometer [22,23,24,25].

2.2 Tutorial on Android Security

The foundation of any operating system security architecture is the security of the device itself. In fixed systems, much of this is based on an assumption of strong physical security such as a locked room or secure enclosure. Since mobile devices are inherently more susceptible to theft and tampering, modern Android devices include several features to secure the device itself. This includes:

- Bootloader locking. Prevents new firmware from being flashed onto the device.
- OEM signature check. Prevents 3rd-party or tampered firmware from being flashed onto the device.
- Secure boot. Requires firmware integrity check at boot time.
- Partition lock. Bootloader prohibits changes to system partition and other sensitive storage areas.
- Read-only mounts. Kernel prohibits changes to system partition and other sensitive storage areas.
- Filesystem encryption. Prevents side-channel access to user data.

To organize an overview tutorial of Android security and its mechanisms, the application life-cycle depicted in Figure 2.1 is used as a road map. The following subsections correspond to the five steps of the app life-cycle.

2.2.1 Development

Android applications are packaged as APK archive files, with the structure shown in Figure 2.2. A key component of this file is the *manifest*, `AndroidManifest.xml`. This is where the developer must declare needed permissions as well as any public interfaces and their associated access control requirements. Most platform resources and actions that can affect the system, other apps,

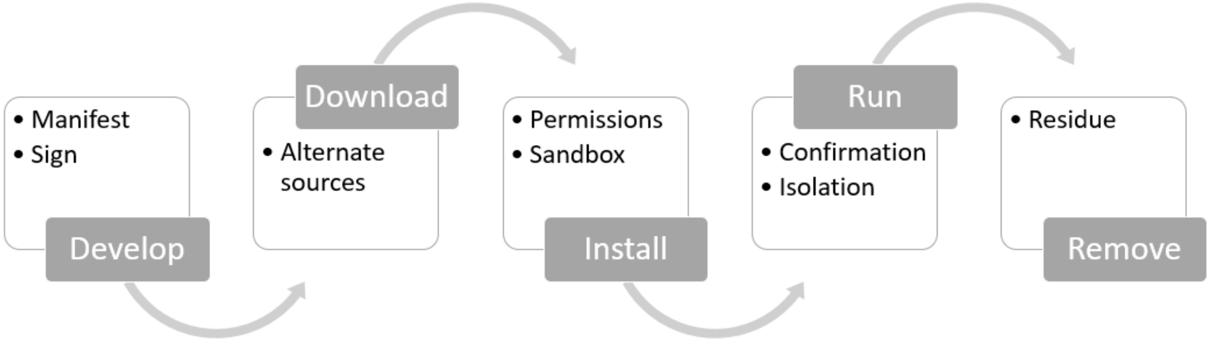


Figure 2.1: Android security mechanisms across the app lifecycle.

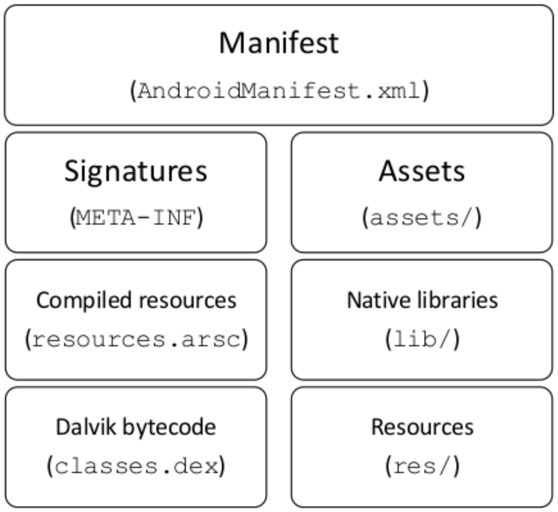


Figure 2.2: APK file structure.

or incur costs require permissions. These include accessing location, Bluetooth, phone, camera, Internet, contacts databases, logs, etc. A full list of permissions allowed in the manifest is found in [26].

Permissions are categorized as *normal*, *dangerous*, *signature* or *signatureOrSystem*. Normal and dangerous permissions are available to all apps, with the latter requiring an explicit approval from the end-user. Signature permissions are only granted to apps signed with the platform key, while signatureOrSystem permissions may be granted to apps that are signed with the platform key or part of the system partition that is defined at system build time.

Uniform Resource Identifier (URI) permissions enable a calling app to grant a callee access to a specific resource, such as a specific file, owned by the calling app. It is also possible for an app developer or system vendor to define custom permissions in the manifest, and use them to implement custom protections for their apps' public interfaces.

All APK files must be digitally signed with the developer's certificate. This allows the author to be identified and is used to establish a security relationship among applications signed with the same certificate. APK signing certificates do not require a certificate authority (CA) and may be self-signed. Also, in order to publish apps in Google's Play store, developers must use certificates that are valid for at least 25 years when registered. This is because any updates to the app must be signed with the same key or the update installation will fail. Unfortunately, there is no standard mechanism for revoking compromised keys, so developers must strictly protect their private keys. In fact, the entire certificate management system of Google Play (or lack thereof) has led to several problems including developers signing all of their apps with a single key, 3rd-party developers signing *thousands* of different customers' apps with a single key, and apps signed with publicly-known private keys [27].

2.2.2 Download

By default, Android is configured to only allow app installation from Google Play. This restriction is part of Google's market-based anti-virus and malware prevention strategy. For the reasons discussed in the example of Section 1.2, 3rd-party anti-virus tools are largely ineffective on stock, unrooted Android devices. To address this, Google scans offerings in the Play store for the presence of malicious libraries and other malware indicators (the exact nature of this scanning algorithm is unknown outside Google). By restricting the installation source to only the Play store, this market-based protection is extended to end-users. Users who opt-in to allow unknown sources bypass this protection and open themselves to significant threats from repackaged and pirated apps. Significant portions of repackaged and pirated apps in these markets have been shown to contain malicious ad libraries, additional functionality, and request more permissions than the originals in Google Play [28].

In order to offer malware protection even to users that install from unknown sources, Google includes a device-based app verification feature starting with version 4.2. Because it is part of the Android system, *Verify apps* does not suffer from the same limitations as 3rd-party anti-virus apps.

2.2.3 Installation

Prior to Android 6.0, permissions were granted to apps at install-time, in an all-or-none fashion. Presented with a partial list (only those permissions categorized as dangerous), users would have to accept all requested permissions or forgo installation. Once accepted, permissions could not be revoked without uninstalling the app.

Beginning with Android 6.0, users do not interact with the permission system at install-time. Instead, permission granting is deferred until a run-time action requires it. At that moment, the user is presented with an allow/deny dialog. In addition, all permissions can be individually granted or revoked at any time via *Settings*.

When an app is installed, it is assigned a Linux *uid* that is used as subject identification in kernel-level access control decisions, just as in standard Linux. This kernel-level identification is the basis for the *application sandbox* that all apps are subject to at run-time. However, unlike standard Linux, in Android the Linux *uid* is separated into fields that have special meanings in the Android Framework. The two most significant digits of the *uid* correspond to the Android *userId*, while the remaining five correspond to Android's *appId*. This allows the same app to run with different Linux *uid*s and *pid*s for different human users or different managed provisioning profiles. In this way, sandbox isolation is maintained among users and profiles, even for the same app used by two users. Chapter 3 contains more detailed specifics about Android's multiuser features and related security mechanisms.

Android's application sandbox is anchored in the process and user isolation afforded by the Linux kernel. From the kernel and Linux userspace point of view, apps are differentiated just as Linux users are on a traditional system. Each app and all of its data files are assigned a unique *uid* and `gid`. For example, a data file for an app assigned *uid* 0010068 would have the following mode, owner and group: `-rwx--- u0_a68 u0_a68`, where `u0_a68` is the Android "username" for *uid* 0010068. Thanks to this kernel-enforced isolation, by default apps cannot read/write files other than their own, manipulate or access other apps, access peripheral devices, access the network, incur costs (e.g., send SMS), or access user data stores (e.g., contacts). One exception are apps whose APKs are signed with the same developer certificate. If so specified in the manifest, these apps are allowed to share the same *uid* and *pid*. This enables developers to share common databases and files among multiple apps in their suites.

To escape the sandbox, apps must have been granted specific permissions by way of the manifest. Certain manifest permissions, such as those controlling access to some hardware resources (e.g., Bluetooth, Internet), logging subsystem, etc., are translated to additional `gid`s. Apps possessing the corresponding permission(s) are then granted membership in the group(s) and the Linux permissions on the corresponding file or device allow group access. For example, the manifest permission `android.permission.INTERNET` is mapped to `gid` 3003 (`inet`), and an app with this permission declared in its manifest is granted membership in this group at install-time. Networking-related sockets are then set with modes, owners, and groups that allow access to this group. These settings are controlled dynamically at boot-time by the `init` process via settings in `init.rc` and `ueventd.rc`, or statically during the system build. For example, the socket `dnsproxyd`, which enables Internet-capable apps to query the Domain Name Service (DNS) via a proxy daemon, has the following mode, owner and group: `srw-rw-- root inet`. This allows the resource to be accessed from within the sandbox directly via system calls, with access control enforced by the kernel. This approach prevents native code from attempting to bypass Framework controls.

Other manifest permissions are checked and enforced by the Android Framework at run-time and are effectively gates that allow different forms of inter-process communication (IPC) between an app and the resources and other apps on the device. In order to enable initial IPC requests, all apps have default permissions to communicate with the kernel's *binder driver*, accessed via `ioctl`s with `/dev/binder`. This character device is owned by `root` but is world-readable and world-writable.

2.2.4 Run-time

Started by `init`, *zygote* is a privileged process which is responsible for launching apps. When an app is launched, zygote forks a new process, sets the assigned *uid* and `gid`(s), and finally drops unneeded privileges and capabilities. Process isolation courtesy of the Linux kernel ensures that an app cannot interfere with other running apps and processes. A running app can attempt to escape its sandbox by two mechanisms as shown in Figure 2.3: Binder IPC (① and ②) and system calls (③).

Binder IPC provides a pathway to Framework resources and components of other apps. Binder itself has no security; instead, access control is enforced at the destination (indicated by ◊ in Figure 2.3) by way of permission checking.

In the case of apps accessing System Services (path ①), the freely accessible binder driver allows access to a directory service, *ServiceManager*. ServiceManager dispatches capabilities known as *handles* for registered System Services such as *LocationManagerService*, *SensorService*, etc. Additional specifics about ServiceManager and its security is included in Chapter 5.

Apps use these capabilities to invoke public methods of the corresponding service via Binder IPC. Any access controls are enforced within each method, usually by checking whether the caller has been granted the necessary manifest permission(s). For example, at the entry point of method `getLastLocation()` in *LocationManagerService*, checks are made to determine whether the

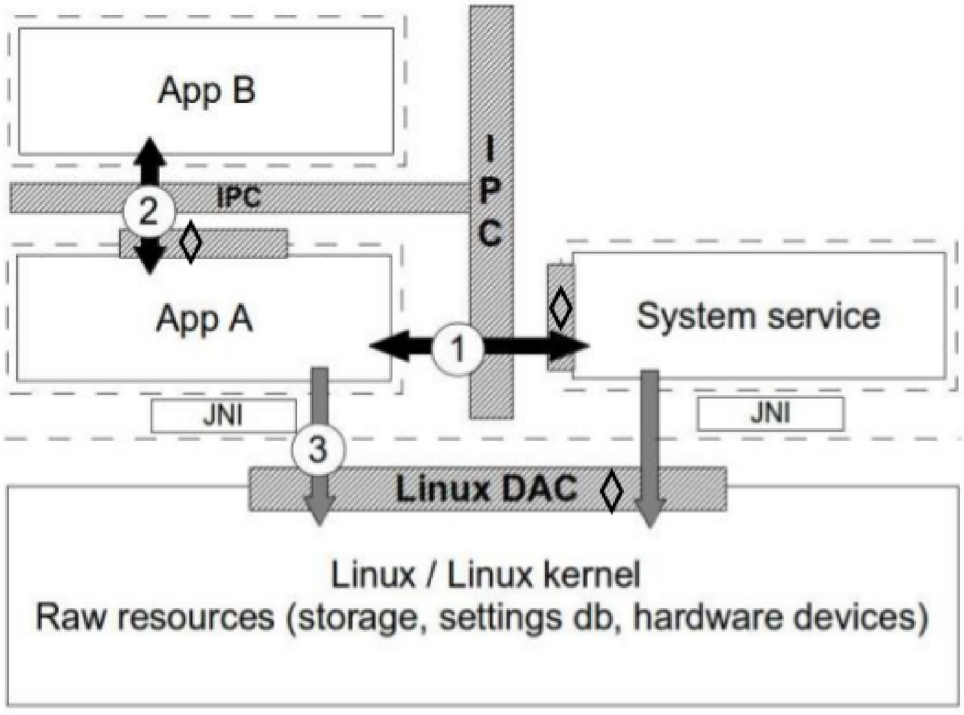


Figure 2.3: Sandboxes (- - -), escape paths (①, ②, ③) and enforcement points (◊).

caller has been granted `android.permission.ACCESS_FINE_LOCATION`, `android.permission.ACCESS_COARSE_LOCATION`, or no permission at all. The result of this check is used to return a most-accurate, less-accurate, or null location, respectively. It is important to note that this access control check is done by the specific method(s) implementing access to the `Location` object, not by a central reference monitor. This pattern of placing access controls as close to resources as possible is a key insight in understanding the security architecture of Android.

As mentioned earlier, Android 6.0 introduces the concept of run-time permission granting. In fact, in this new model, no permissions are granted at install time. When the app is launched and encounters a situation which requires use of a dangerous permission declared in the manifest, the user is prompted. By choosing *allow* at this prompt, the app is permanently granted the permission, which must then be manually revoked via *Settings*.

2.2.5 Removal

When apps are uninstalled, the assigned *uid* is freed up for reuse and its data directories are removed. In some cases, remnants of the installation persist, such as credentials, capabilities, settings, history, etc., and may cause security problems [29]. This continues to be an active research area.

Chapter 3

Android Access Control Evaluation Methodology

The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards.

- Gene Spafford

The Android operating system is rapidly evolving, with major releases now occurring approximately every six months. Each of these releases contain new features, enhancements, and refinements that have significant security implications. These implications must be understood, especially when devices are employed for critical and sensitive applications or in adversarial environments. Unfortunately, Android is a very complex system, and as explained in Chapter 2, its security architecture has been designed as an open system whereby access controls are placed as close to the corresponding resource as possible. In this architecture, access control policy and implementation are the responsibility of the resource itself. While this modular approach makes adding new or unique resources easier, evaluating the soundness of platform-wide changes as they relate to every resource becomes very difficult because of the many access control points involved.

This chapter introduces a methodology for modeling, discovering and evaluating access controls in Android. It is especially well-suited to evaluating how platform-wide architectural changes or feature additions impact the security of the many separate resources whose access controls may have been designed under earlier, potentially invalidated assumptions. The chapter also describes the results from a case study which demonstrates the effectiveness of the method. The case study applied the method to evaluating the security of Android's multi-user framework, a significant platform change which was first introduced in the 2012-2013 time-frame. The multi-user extensions to Android represent exactly the kind of platform change that results in security problems involving resources previously designed under assumptions invalidated by the change.

3.1 Introduction

The impressive growth of Android includes not only the number of traditional smartphone devices running it, but an expansion to other "keyboardless" and embedded devices such as tablets, home entertainment equipment, automobile dashboards, and home appliances. While the typical smartphone is a personal single-user devices, many of the others are intended for use by several individuals. Thus, one major driving force behind the features included in recent releases of Android is the need for multi-user support, which has grown along with its expansion from strictly personal devices to those with varied purposes in multi-user environments. Today, the vast majority of fielded Android devices are multi-user capable, with Google Dashboards reporting in June 2016 that 89.2% of active devices were running a multi-user-capable version [30] (up from 27% in 2014).

Multi-user features take two different forms in Android: *Multiple Users* (MU), introduced in version 4.2 (API 17) in November 2012, and *Restricted Profiles* (RP), introduced in version 4.3 (API 18) in July 2013. With each subsequent release, the multi-user functionality was refined and expanded to fit other contexts such as corporate environments.[1] Targeted towards sharable devices such as

[1] Since the time of this study, *Managed Provisioning* was introduced in Android version 5.0 (API 21) to allow enterprise-

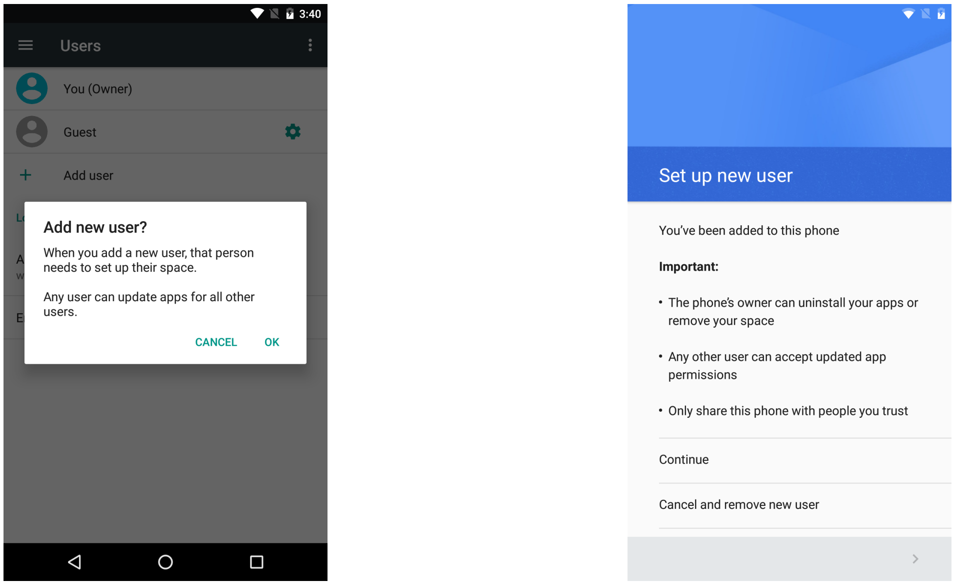
tablets, these enhancements strive to provide individual, isolated user spaces on a single physical device. Each user space supports a separate set of accounts, apps, settings, files, and user data, distinct from those of the primary owner [32], however there are slight differences in the functionality they provide.

Multiple Users (MU) designates the main account as *Owner*. Through the device settings, the owner account may create additional MU accounts. These secondary accounts are essentially the same as the owner, except for the fact that they cannot manage (i.e., create, modify, delete) other users. MU accounts enjoy most of the other privileges and functionality of the owner, including managing the device's wireless and network settings, pairing Bluetooth devices, customizing sound and display settings, installing/removing their own apps, adjusting privacy settings (e.g., location access), and configuring security features (e.g., screen lock, credentials). Each account also has a separate virtual SD card storage area within the physical SD card.

Restricted Profiles (RP) are similar to MU accounts, but they lack several key functionalities compared to owner and MU accounts. Like MU accounts, RP accounts cannot manage users. In addition, RP accounts are restricted from installing apps. Instead, the owner account "turns on" specific apps from the set of installed apps for the RP account.

A reasonable layperson might assume that these enhancements would provide user and profile isolation similar to that provided by today's desktop multi-user systems. However, end-users working with multi-user features immediately encounter warning signs that this may not be the case. When creating new users on a device, two telling confirmation dialogs appear as shown in Figure 3.1. The dialog shown in Figure 3.1a warns that "any user can update apps for all other users", while that of Figure 3.1b implores the new user to "only share this phone with people you trust".

managed apps to coexist with personal apps in the same user account and launcher. My own brief investigation [31] revealed that the underlying technical implementation of Managed Provisioning is very similar to that described here for Multiple Users and Restricted Profiles. As such, many of the security concerns outlined here are applicable to Managed Provisioning.



Add new user?

When you add a new user, that person needs to set up their space.

Any user can update apps for all other users.

(a) Warning to device owner.

You've been added to this phone

Important:

- The phone's owner can uninstall your apps or remove your space
- Any other user can accept updated app permissions
- Only share this phone with people you trust

(b) Warning to new user.

Figure 3.1: Warning messages encountered while creating additional user accounts.

To security-minded individuals, these kinds of warnings raise red flags. Combined with the fact that multi-user features were basically "bolted on" to an existing design, and the obvious security implications of multi-user features for *any* system, there is plenty of reason to worry about the soundness of these new capabilities. In fact, Android's evolution towards a multi-user system is not unlike that which happened with Microsoft's Windows operating systems in the 1990s, which started out as a single-user system. Compared to Unix, which was designed from the outset with multiple users in mind, Windows' later inclusion of these features was much more problematic in terms of security [33]. Thus, is important to assess whether Android's new multi-user framework accounts for the fact that the single-user assumption of the original design has been invalidated, as Microsoft arguably failed to do with initial multi-user versions of Windows.

However, rather than conducting an evaluation specific to the multi-user framework, this work contributes a **systematic access control evaluation *methodology*** that is tailored to the unique characteristics of Android, but generic enough to be suitable for use in evaluating any aspect of the system's access controls. The method enables one to gain security (and vulnerability) insights, which then lead to hypotheses about potential security problems. These hypotheses can be tested by way of focused experimentation. The efficacy of the method is proven via a case study on Android's multi-user framework, a recent, security-related addition to the system, which had not been studied before.[2]

3.1.1 Threat Model

Most security evaluations focus their scope by defining a threat model. In fact, as stated earlier, Google recommends sharing multi-user devices only with trustworthy people. Unfortunately, varying definitions of trust, different expectations for security and privacy, and a wide variety of use cases make this a very ambiguous statement. Because of this, and since the methodology is intentionally generic with respect to the types of Android access controls being evaluated, we do not immediately identify a specific threat or scenario. Instead, insights and knowledge are produced first, through a systematic, exhaustive analysis, independent of any particular threat mindset. The hypotheses which can then be generated factor in the specific threat scenarios.

The rest of the chapter is organized as follows: a background on how multi-user has been implemented in Android is first provided in Section 3.2 so that later descriptions of the methodology can include the multi-user case study context for clarity and as an example. Next, Section 3.3 describes the systematic methodology for gaining insights into relevant access controls and hypothesizing about potential vulnerabilities. A sampling of the most interesting hypotheses are given in Section 3.3.3. Finally, Section 3.4 presents the findings from experiments designed to test the hypotheses. Related work is consolidated in Chapter 6.

[2] A high confidence anonymous reviewer of [34], the peer-reviewed publication describing this work, stated that the paper is "...the first work that I'm aware of that looks at the security of Android's multi-user framework."

3.2 Background

Before describing our investigation, a brief technical overview of Android's implementation of multiple users is needed. The following section expands on the general background material of Chapter 2, and is divided into four subsections: Android framework extensions, filesystem configuration, kernel mechanisms, and run-time considerations. Each subsection may contain a brief security discussion in order to emphasize aspects that are important for later discussions. Throughout the chapter, the Linux user ID and group ID are referred to as *uid* and *gid*, respectively, while IDs within the Android framework are denoted by *userId*, and *appId*.

3.2.1 Framework - *userId*

Version 4.2 added **`android.os.UserHandle`** class to represent multiple users on the device. This class designates *userId* 0 as the device owner, and several special *userId*s to represent all users (*userId* -1), the current user (*userId* -2), the current user or self (*userId* -3) and the null user (*userId* -10000). Actual *userId*s are assigned by the *UserManagerService* class (also introduced in 4.2) when new users or new restricted profiles are created by the device owner. This class defines the starting *userId* as 10, and increments it by 1 every time a new user or profile is created, until the number of current users equals the maximum number defined by the system property *fw.max_users* from **`build.prop`**. State is maintained in **`/data/system/users/userlist.xml`**, where a list of currently-assigned users and the next available *userId* is stored. On devices where users or profiles have been repeatedly added and deleted, the *userId*s may be re-used, although the next available counter continues to increment as shown by the example of Listing 3.3. *userId*s are assigned in the same way regardless if they are for a secondary user or a restricted profile.

For example, Listing 3.1 shows a representative **`userlist.xml`** file from a fresh device with one secondary user and one restricted profile added. After deleting these two accounts and adding a third secondary user, the file appears as shown in Listing 3.2.

Listing 3.1: ***userlist.xml*** *showing users 0 (owner), 10, 11, and next available userId 12.*

```
<?xml version=’1.0’ encoding=’utf-8’ standalone=’yes’ ?>
<users nextSerialNumber="12" version="4">
    <user id="0" />
    <user id="10" />
    <user id="11" />
</users>
```

Listing 3.2: ***userlist.xml*** *after deleting users 10, 11, and adding a user, showing users 0, 12 and next available userId 13.*

```
<?xml version=’1.0’ encoding=’utf-8’ standalone=’yes’ ?>
<users nextSerialNumber="13" version="4">
    <user id="0" />
    <user id="12" />
</users>
```

Listing 3.3: ***userlist.xml*** *after deleting users 10, 11, and adding a user, showing users 0, 12 and next available userId 13.*

```
<?xml version=’1.0’ encoding=’utf-8’ standalone=’yes’ ?>
<users nextSerialNumber="27" version="4">
    <user id="0" />
    <user id="11" />
</users>
```

As has always been the case in Android, each installed application is assigned an *appId*.[3] `android.os.Process` class defines ranges of *appId*s that can be assigned to different types of apps. Normally, these IDs range from 10000 to 99999.

In the past, *appId*s were the same as the Linux *uid*, thus enabling process, memory and filesystem isolation among the different apps installed on a device. With the advent of the multi-user framework, the most significant bits of the Linux *uid* take on the semantics corresponding to *userId*, while the remaining bits continue to correspond to *appId*. Specifically, the Linux *uid* is obtained by concatenating the *userId* and *appId* as follows:

[3] Before the introduction of multi-user, *uid* and *userId* were used interchangeably to refer to the unique identifier for each app installed on the system. In versions with multi-user extensions, *userId* is used to denote the actual user, while *appId* is the designation for each app’s unique ID. However, there are still several instances of code and files that use *userId* to refer to apps. For example, the *sharedUserId* tag in `AndroidManifest.xml` actually refers to package names which will share the same *appId*.

$$\mathit{uid} = \mathit{userId} \times \mathit{PER\_USER\_RANGE} + (\mathit{appId} \bmod \mathit{PER\_USER\_RANGE})\,, \tag{3.1}$$

where the default *PER_USER_RANGE* is 100000.

Likewise, *userId* and *appId* can be recovered from *uid* using

$$\mathit{userId} = \mathit{uid} \ \mathrm{div} \ \mathit{PER\_USER\_RANGE} \tag{3.2}$$

and

$$\mathit{appId} = \mathit{uid} \bmod \mathit{PER\_USER\_RANGE}\,, \tag{3.3}$$

where *div* denotes integer division and *mod* the modulo operation. The `UserHandle` class includes methods `getUid`, `getUserId`, and `getAppId` for performing these conversions.

Thus, the Linux *uid* is comprised of a two-digit Android *userId* (00, 10, 11, 12, ...) concatenated with a five-digit Android *appId* (10000, 10001, ...). For example, an app with *appId* 10056 will run with Linux *uid* 0010056 when started by the owner (*userId* 0), and Linux *uid* 1010056 when started by the first secondary user or restricted profile (*userId* 10).

System *uid*s not directly associated with apps are still in the range 0-9999. For example, `root` is *uid* 0, `system` is *uid* 1000, `radio` is *uid* 1001, and `shell` is *uid* 2000.

3.2.2 Framework - Permissions

Several new permissions have been introduced with the advent of multi-user support. These include `MANAGE_USERS`, `INTERACT_ACROSS_USERS` and `INTERACT_ACROSS_USERS_FULL`, which are used to protect some types of inter-user functionalities such as `startActivityAsUser()`.

Generally, checks for these permissions are bypassed if the calling process has a **root** or **system** *uid*. Several are also bypassed for processes running as **shell**. As *signatureOrSystem* permissions, they will not be granted to apps not in the **/system** partition or signed with the platform key, and thus are not obtainable by 3rd-party apps.

3.2.3 Framework - Package Management

To accommodate multiple users, Android's package management system was modified so that secondary users can choose different sets of installed apps, and the owner can choose which apps are enabled or disabled for RPs. However, as currently implemented, package management is still largely *platform*-centric rather than *user*-centric. Although it may *appear* that each user has their own independent set of apps installed, in reality, each app is installed *once* for the entire platform, and then either enabled or disabled for each user. Evidence of this is seen in the fact that a device with multiple users still has only one, platform-wide **packages.list** file to map package names to corresponding data directories, *appId*s, and *gid*s. This can be seen in the example **packages.list** file shown in Listing 3.4. This file is from a multi-user device with two additional accounts configured. For example, the package **jackpal.androidterm** shown in Listing 3.4 was installed by the secondary user and is not visible at all to the owner or other users. However, **packages.list** only contains the package name, *appId*, flag, data directory path, signing key name, and *gid* assignments.

Listing 3.4: ***packages.list*** *excerpt showing the lack of user-specific information.*

```
com.android.soundrecorder 10051 0 /data/data/com.android.soundrecorder release 3003,1028,1015
com.android.voicedialer 10014 0 /data/data/com.android.voicedialer release 3002
com.android.defcontainer 10003 0 /data/data/com.android.defcontainer platform 1028,1015,1023,2001,1035
com.android.launcher 10008 0 /data/data/com.android.launcher shared none
com.android.quicksearchbox 10050 0 /data/data/com.android.quicksearchbox shared 3003
com.android.contacts 10002 0 /data/data/com.android.contacts shared 3003,1028,1015
com.android.inputmethod.latin 10035 0 /data/data/com.android.inputmethod.latin shared 1028,1015
com.android.phone 1001 0 /data/data/com.android.phone platform 3002,3001,3003,1028,1015
com.android.calculator2 10020 0 /data/data/com.android.calculator2 release none
com.android.proxyhandler 10012 0 /data/data/com.android.proxyhandler platform 3003
com.android.htmlviewer 10033 0 /data/data/com.android.htmlviewer release 1028
com.android.providers.calendar 10001 0 /data/data/com.android.providers.calendar release
    3003,1028,1015
com.android.bluetooth 1002 0 /data/data/com.android.bluetooth platform
    3003,3002,3001,1028,1015,3005,1016,3008
jackpal.androidterm 10058 0 /data/data/jackpal.androidterm default 3003,1028,1015
```

Moreover, there is also only one platform-wide **`packages.xml`** file for associating package names and *appId*s with signature keys, native library paths, code paths, granted permission(s), and special conditions such as *sharedUserIds*. As illustrated by the **`packages.xml`** excerpts in Listing 3.5, the file has four major sections: a **`<permissions>`** block where permissions for the entire platform are defined; one or more **`<package>`** blocks where each installed package is associated with its signature identifier, code path, native library path, **`ApplicationInfo`** flags, permissions, *appId* (shown in file as *userId* for historical reasons explained earlier), and other package-specific settings; one or more **`<shared-user>`** blocks where shared *appId*s (indicated by *sharedUserId*) are associated with permissions; and a **`<keyset-settings>`** block where signature identifiers are mapped to the actual public key.

*Listing 3.5: **`packages.xml`** excerpts showing the lack of user-specific information.*

```
<packages>
    <last-platform-version internal="19" external="19" />
    <permission-trees />
    <permissions>
        <item name="android.permission.CHANGE_WIFI_MULTICAST_STATE" package="android"
            protection="1" />
        <item name="android.permission.WRITE_CALL_LOG" package="android" protection="1" />
        <item name="android.permission.CLEAR_APP_CACHE" package="android" protection="1" />
        <item name="android.permission.AUTHENTICATE_ACCOUNTS" package="android" protection="
            1" />
        <item name="android.permission.ACCESS_WIMAX_STATE" package="android" />
        <item name="android.permission.ASEC_ACCESS" package="android" protection="2" />
        <item name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS" package="android
            " protection="1" />
        <item name="android.permission.INTERNAL_SYSTEM_WINDOW" package="android" protection=
            "2" />
        <item name="android.permission.CAMERA_DISABLE_TRANSMIT_LED" package="android"
            protection="18" />
        <item name="android.permission.ACCESS_MOCK_LOCATION" package="android" protection="1
            " />
        <item name="android.permission.ACCESS_NETWORK_STATE" package="android" />
        <item name="android.permission.CHANGE_BACKGROUND_DATA_SETTING" package="android"
            protection="2" />
        <item name="android.permission.GET_DETAILED_TASKS" package="android" protection="2"
            />
        ...
    </permissions>
    <package name="com.android.contacts" codePath="/system/priv-app/Contacts.apk"
        nativeLibraryPath="/data/app-lib/Contacts" flags="1078509125" ft="155316fb3a0" it="
        155316fb3a0" ut="155316fb3a0" version="19" sharedUserId="10002">
        <sigs count="1">
            <cert index="2" />
        </sigs>
        <signing-keyset identifier="3" />
    </package>
    <package name="com.android.providers.userdictionary" codePath="/system/app/
        UserDictionaryProvider.apk" nativeLibraryPath="/data/app-lib/UserDictionaryProvider"
         flags="572933" ft="155316d2f18" it="155316d2f18" ut="155316d2f18" version="19"
        sharedUserId="10002">
        <sigs count="1">
            <cert index="2" />
        </sigs>
        <signing-keyset identifier="3" />
    </package>
    <package name="jackpal.androidterm" codePath="/data/app/jackpal.androidterm-1.apk"
        nativeLibraryPath="/data/app-lib/jackpal.androidterm-1" flags="572996" ft="1567
        f5e9058" it="1567f5e928b" ut="1567f5e928b" version="63" userId="10058">
        <sigs count="1">
            <cert index="4" key="30820242308201
                ab02044adb879a300d06092a864886f70d01010405003067310b30090603550406130275733
                ..." />
        </sigs>
        <perms>
            <item name="android.permission.READ_EXTERNAL_STORAGE" />
            <item name="android.permission.WAKE_LOCK" />
            <item name="android.permission.ACCESS_SUPERUSER" />
            <item name="android.permission.WRITE_EXTERNAL_STORAGE" />
            <item name="android.permission.INTERNET" />
        </perms>
        <signing-keyset identifier="1" />
    </package>
    ...
    <shared-user name="android.uid.shared" userId="10002">
        <sigs count="1">
            <cert index="2" />
        </sigs>
        <perms>
            <item name="android.permission.READ_EXTERNAL_STORAGE" />
            <item name="android.permission.WRITE_EXTERNAL_STORAGE" />
            <item name="android.permission.WRITE_CALL_LOG" />
            <item name="android.permission.REBOOT" />
            <item name="android.permission.READ_SOCIAL_STREAM" />
            <item name="android.permission.ACCESS_COARSE_LOCATION" />
            <item name="android.permission.READ_CONTACTS" />
            <item name="android.permission.GET_ACCOUNTS" />
            <item name="android.permission.WRITE_CONTACTS" />
            <item name="android.permission.READ_PHONE_STATE" />
            ...
        </perms>
    </shared-user>
    ...
    <keyset-settings>
        <keys>
            <public-key identifier="1" value="
                MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnrCftbkq1488N8niH5NFCL5XJ/Fb..." />
            ...
        </keys>
        <keysets>
            <keyset identifier="1">
                <key-id identifier="1" />
                ...
        </keysets>
        <lastIssuedKeyId value="1" />
        <lastIssuedKeySetId value="0" />
    </keyset-settings>
</packages>
```

Notably, this file also associates the aggregate permission list for `<shared-user>` packages to *appId* without regards to any particular framework *userId*. For example, the packages `com.android.contacts` and `com.android.providers.userdictionary` each declare *sharedUserId*s of 10002 which means they are each granted all of the permissions defined by shared user `android.uid.shared`.

Because the content and structure of the above files was not changed when multi-user features were added to Android, *PackageManager* must have a way of keeping track of each user's specific app installation status. This is accomplished by way of individual `package-restrictions.xml` files for each user (default location: `/data/system/users/<userId>/`). When an individual user installs an app for themselves, the app is really installed on the platform (as indicated by the `platform.list` and `platform.xml` files, and then simply "hidden" from other users by tagging the package name with `inst=false` in that user's `package-restrictions.xml`. In the `jackpal.androidterm` example used earlier, the `package-restrictions.xml` file for the user who installed the package contains

`<pkg name="jackpal.androidterm" stopped="true" nl="true"/>`,

while the same file for all other users and profiles contains

`<pkg name="jackpal.androidterm" inst="false" stopped="true" nl="true"/>`.

We confirmed that the `inst="false"` parameter controls app visibility to users by manually editing the file to remove it and observing that the app becomes available.

3.2.4 Filesystem

To support multi-user, several changes to the filesystem organization were made. Whereas the single user's app data was previously stored under `/data/data/<package_name>`, this data is now isolated for each user under `/data/user/<userId>/<package>`. To maintain backwards compatibility, the owner's (*userId* 0) app data is still stored under `/data/data/<package>`, with a symbolic link from `/data/user/0` to `/data/data`. Subdirectories in these locations are owned by the *uid* for the respective user and app. Strong isolation is achieved through the use of Linux bind mounts and filesystem namespaces [35].

3.2.5 Kernel

Since Linux is naturally a multi-user system, implementing Android's multi-user extensions at the kernel level did not require any changes to the kernel itself. For all versions of the Linux kernel used in Android, the Linux *uid* is an unsigned 32-bit integer which can represent over 4 billion unique *uid*s. Thus, the *uid* discussed above, formed from the Android *userId* and *appId*, uniquely identifies both the user and app, and is directly used as the Linux *uid*. In this way, standard Linux discretionary access control (DAC) can provide isolation not only among apps, but also among each user's data files for a particular app.

3.2.6 Run-time

On a running device, only one user can be "logged in" at any one time. However, through the switch users function, multiple users introduced the concept of the *current user*, which refers to the user interacting with the device. We refer to other users who may have been using the device before it was switched to the current user as "inactive users." Although these users cannot interact with the device, many of the underlying processes associated with their active session, are left running, as shown in Listing 3.6 where app processes from both the owner (*uid*s beginning with `u0_`) and a secondary user (those beginning with `u10_`) are running while the owner is using the device. The inactive user's apps are paused and their background services may be left to run. On builds we have used, there is a limit of 3 to the number of users that can be inactive before their processes are completely removed.

Listing 3.6: Partial process listing from a multi-user device showing processes from two users running simultaneously.

```
...
root      224   1     552688 57232 ffffffff b7676770 S zygote
media     225   1     78740  23504 ffffffff b75f4586 S /system/bin/mediaserver
system    551   224   650404 58416 ffffffff b767807b S system_server
u0_a7     607   224   602888 68144 ffffffff b767807b S com.android.systemui
radio     713   224   583828 40852 ffffffff b767807b S com.android.phone
u0_a8     726   224   615000 52292 ffffffff b767807b S com.android.launcher
u0_a55    791   224   560552 30312 ffffffff b767807b S com.android.smspush
u0_a40    805   224   562260 32440 ffffffff b767807b S com.android.music
wifi      883   1     5424   2156  c021a64f b75f6770 S /system/bin/wpa_supplicant
dhcp      1146  1     1620   488   c021a64f b76da146 S /system/bin/dhcpcd
u10_a8    1239  224   592460 64512 ffffffff b767807b S com.android.launcher
u10_a7    1269  224   566924 42024 ffffffff b767807b S com.android.systemui
u10_a5    1358  224   565424 38488 ffffffff b767807b S android.process.media
u10_a25   1515  224   565728 35796 ffffffff b767807b S com.android.deskclock
u10_a29   1532  224   572716 37892 ffffffff b767807b S com.android.email
u10_a40   1585  224   562260 32432 ffffffff b767807b S com.android.music
system    1598  224   572196 33272 ffffffff b767807b S com.android.settings
u10_system 1614  224   572196 33276 ffffffff b767807b S com.android.settings
u10_a32   1636  224   605964 41540 ffffffff b767807b S com.android.gallery3d
u0_a35    1669  224   570692 38484 ffffffff b767807b S com.android.inputmethod.latin
u0_a48    1682  224   561888 31144 ffffffff b767807b S com.android.printspooler
u0_a2     1702  224   563436 36836 ffffffff b767807b S android.process.acore
u0_a1     1732  224   565668 35076 ffffffff b767807b S com.android.providers.calendar
u10_a1    1748  224   563640 35132 ffffffff b767807b S com.android.providers.calendar
u0_a21    1904  224   573064 33828 ffffffff b767807b S com.android.calendar
u10_a9    1921  224   565712 34396 ffffffff b767807b S com.android.mms
u10_radio 1943  224   563976 32948 ffffffff b767807b S com.android.phone
u10_a21   1958  224   569984 34872 ffffffff b767807b S com.android.calendar
u0_a5     1995  224   564904 34964 ffffffff b767807b S android.process.media
u0_a9     2011  224   567784 34968 ffffffff b767807b S com.android.mms
u0_a25    2046  224   563640 34508 ffffffff b767807b S com.android.deskclock
```

3.3 Method and Model

At the core of any security investigation lies the question of whether the system design is based on valid assumptions. As Android evolves into a multi-user system, what once may have been a set of valid assumptions may suddenly be undermined by emerging system characteristics and/or usage models. In the case of the multi-user framework, the original assumption of a benign, single-user environment is no longer valid. Rather than a single owner who has administrative authority over most aspects of system configuration and would not attack or intentionally mis-configure his own system, there is now an environment where it is plausible for secondary users to bypass restrictions, attack other users, or deliberately reconfigure the system in an unauthorized way. This insight gives us a way to focus the scope of the investigation.

3.3.1 Scope

To define the scope of this evaluation, we consider the overall architecture of Android depicted by Yaghmour's [36] high-level architectural view shown in Figure 3.2. This diagram is not specific to multi-user, which enables us to employ the methodology below to other types of access control evaluations besides the multi-user case study discussed here. This diagram shows broad categories of system resources such as stock apps and system services, which become the subject and object categories. The goal is to enumerate all relevant subject-object combinations and then evaluate the suitability of the access control path(s) between them, where suitability is dependent on whatever case is being looked at. In the case study presented here, our scope is limited to scenarios whereby a *secondary* user exercises all possible paths to access resources and/or gain privileges.

3.3.1.1 Identify subjects

With the above in mind, the subjects considered are apps and user interface (UI) elements that the users can install or use. This list includes user-installed as well as stock apps (e.g., Settings), with the key difference being privilege (i.e., stock apps can have `signatureOrSystem` permissions while user-installed apps cannot). In Figure 3.2, these are labeled as SUBJECT A for the case of a user-installed app making API requests to other parts of the system, and SUBJECT B for cases whereby the user interacts with stock apps to access resources.

For the example multi-user case, the subjects we identify are restricted to those apps that a *secondary user* can install or use. A very important difference about the secondary user subjects considered compared to those of the single-user case is that the secondary user who launched the app may not necessarily still be the current user of the device. For example, components of an

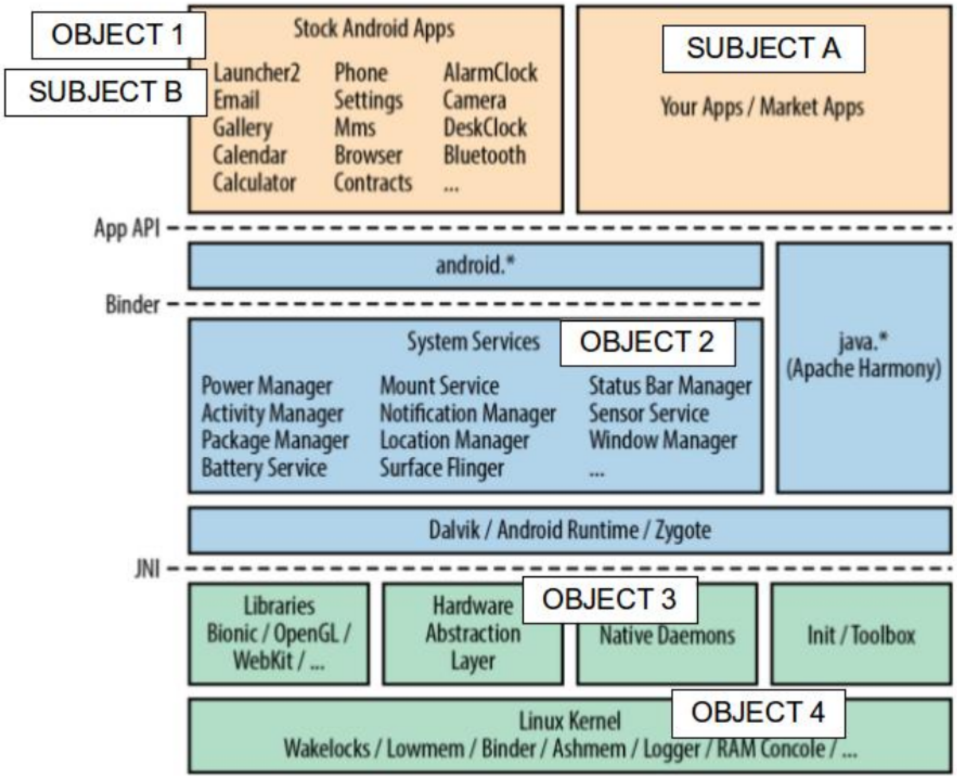


Figure 3.2: Investigation problem space showing various subject-object combinations. Adapted from [36] with the permission of O'Reilly Media, Inc.

app launched by a secondary user prior to switching to another user may still function.

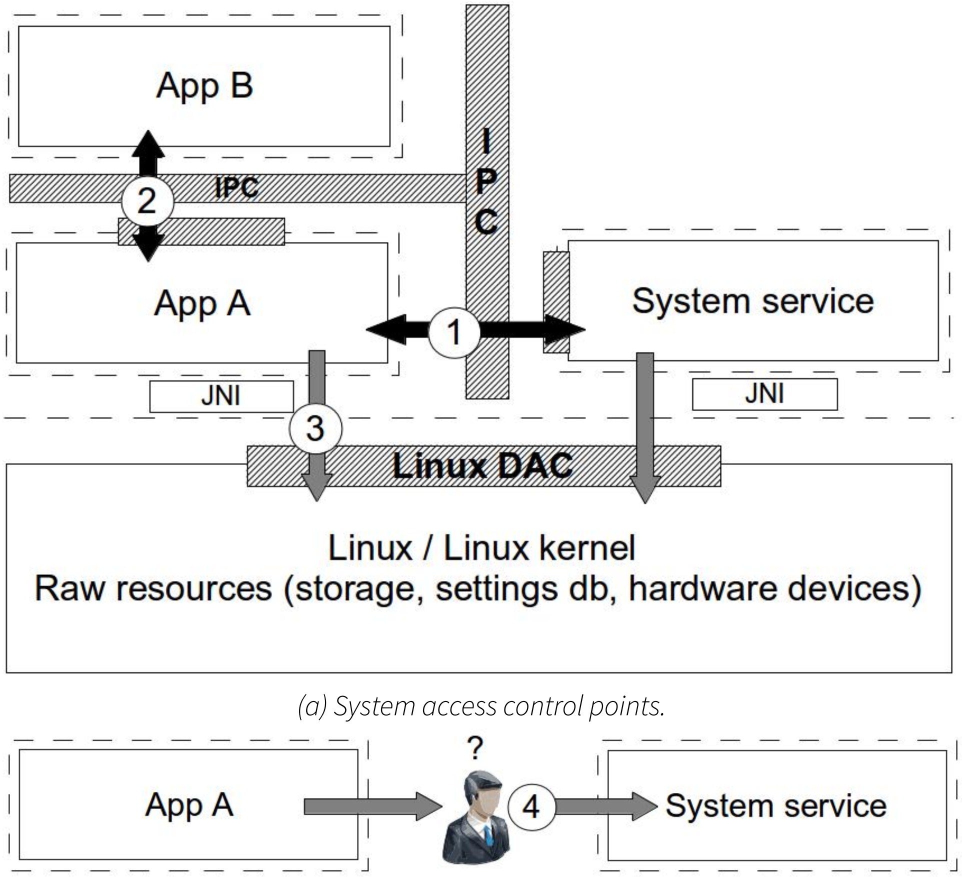
3.3.1.2 Identify objects

The resources available to the subjects discussed above are represented as the object in the access control investigation. Examples from Figure 3.2 include public interfaces of other apps (OBJECT 1), services (OBJECT 2), abstracted hardware devices (OBJECT 3) and kernel objects (OBJECT 4).

The most important difference about the objects considered for the multi-user case study compared to the single-user case is that some resources may be shared with other users on the device. For example, hardware devices such as the camera, or common settings databases are objects that are shared among multiple users on the device.

3.3.1.3 Identify access control paths

Between these subjects and objects are communication paths that may include access control mechanisms. We draw upon the work of [10], [20], and [37] to present the simplified models of Figure 3.3 that show communication paths pertinent to our investigation. This simplified model gives us an Android-specific frame of reference with which we can consider each SUBJECT-OBJECT path's adequacy for whatever aspect of access control is under investigation.



(a) System access control points.

(b) Person-based access control.

Figure 3.3: Android communication paths showing access control points.

Figure 3.3a depicts each app and system service contained by separate sandboxes as indicated by the dotted lines around them. Communication among these sandboxes (denoted by bi-directional arrows) is done through Intents and Binders. Using Binders, apps obtain access to services or providers (path ①), and using Intents may launch exported activities of other apps (path ②). These paths include access control points provided either by the system (e.g., as part of the Intent or Binder mechanism), or at the public interface of the object itself. Using the native interface, apps may also make system calls to directly request resources controlled by the Linux kernel (path ③), and these are subject to Linux DAC. These three paths are permission-based, access control list (ACL)-based, or based on some combination of these.

Figure 3.3b shows another communication path with access control typical of smartphones and tablets, that performed by the user (path ④). In this case, the current user makes the decision to allow or disallow access to a resource such as location, for instance. We refer to this as *person-based* access control to avoid confusion with the notion of users on the device.

A fifth type of path, not shown, are those that have *no* access control along them.

3.3.2 Questions & Insights

As we study the inner-workings of Android's multi-user features, we are able to make two observations. First, the new features have introduced important new considerations for the subjects and objects shown in Figure 3.2. Examples of this include the concept of apps run by a *userId* that is different than that of the current user, and person-based access control decisions being made by multiple individuals. Second, even though none of the access control paths of Figure 3.3 are unique or dedicated to the extensions, some have been modified to account for the presence of multiple users on the device. Examples include methods that include checks for `INTERACT_ACROSS_USERS` permission and apps that express different versions of their UI to restricted users than they do to the device owner.

These observations lead us to the following top-level questions for our investigation:

1. Do Android's access control points properly account for the new considerations regarding subjects and objects?
2. If not, can a secondary user exploit these shortcomings, and what is the potential damage?

In order to answer these questions, we enumerate all of the meaningful subject-object combinations within the broad categories identified by Figure 3.2, and identify the corresponding access control paths from Figure 3.3. This gives us a comprehensive list of specific things to study. For example, a user-installed app (SUBJECT A) can send an Intent using `startActivity()` to launch any exported activity of any other app (OBJECT 1). Thus, we study the system's Intent mechanism and the specifics of how these activities are exported. Specifically, we examine the source code in order to determine what considerations, if any, do the Intent mechanisms and exported activities give to multiple users. If none or partial, we consider whether there should be protections and how a secondary user might exploit the shortcomings.

3.3.3 Hypotheses About Multi-User Security

This last step allows us to develop a set of hypotheses which can be used to design experiments for testing the adequacy of access controls and demonstrating the consequences. Because they must be testable, the hypotheses may include additional details about the threat model or scenario required to exploit the potential vulnerability. We present a partial list of our most interesting hypotheses for the multi-user case study here:

Hypothesis 1: *Secondary users may be able to bypass their restrictions by exploiting the unprotected public interfaces of system apps.* Secondary users are supposed to lack certain capabilities that the owner has, such as mobile plan settings. However, from our study of how access control restrictions are implemented in Settings, we see that many are accomplished by way of hiding portions of the UI, while the corresponding activities are exported publicly. This situation corresponds to a particular OBJECT 1 in Figure 3.2 (Settings) that is shared among all users without adequate access control along path ② of Figure 3.3a. Results from testing this hypothesis are contained in Section 3.4.1.

Hypothesis 2: *Secondary users may be able to maliciously reconfigure critical platform-wide settings that are persistent across user switches.* Secondary users possess certain administrative capabilities (e.g., network settings) that are normally reserved for privileged users on mature multi-user systems such as Linux. Under Android's single user assumption, some of these settings are protected by person-based access control since UI interaction by the benign user is required to prevent malicious apps from making invisible changes programmatically. However, when the benign user assumption is invalid, Figure 3.2's shared resources protected only by Figure 3.3b's person-based access controls (path ④) can be maliciously manipulated. The consequences are even more severe for cases where the configurations are persistent across user switches, such as in the case of network configuration. Results from testing this hypothesis are contained in Section 3.4.2.

Hypothesis 3: *Inactive users may be able to spy on active users by exploiting improper access control enforcement on shared hardware resources.* As mentioned above, multi-user extensions introduce the concept of current and background users. However, unlike a true multi-user system such as Linux, which generally allows multiple remote logins simultaneously, there can only be one active user "logged in" an Android tablet at any one time. However, our enumeration of Figure 3.2 objects discovered apps and services that have access to shared objects and are allowed to continue operations even after a user switch occurs. Of these, certain ones such as audio, camera and location have obvious privacy implications if used without the current user's knowledge or consent. From an analysis of the implicated access control paths ① and ④ in Figure 3.3, we find that authorizations granted when the secondary user is the current user may not be properly reconsidered after a user switch. Results from testing this hypothesis are contained in Section 3.4.4.

Hypothesis 4: `sharedUserId` *permissions may not be properly separated when* `sharedUserId` *apps are installed by different users.* Multiple users extensions bring with them the idea that each user may have different settings, preferences, and apps. Obviously, these should be isolated such that one user cannot accidentally inherit permissions or capabilities from another. Because our enumeration of all subject and object combinations of Figure 3.2

included apps that leverage the **sharedUserId** feature, we discovered problematic situations with overprivilege that can occur when different users install apps with **sharedUserId**s. In particular, we see that access controls at Figure 3.3a locations ①, ②, and ③ fail to differentiate the subject because each **sharedUserId** app's permissions are commingled with others of the same *appId* in a single **packages.xml** file shared among all users. Results from testing this hypothesis are contained in Section 3.4.3.

Hypothesis 5: *A malicious user may be able to exploit the shared package management system to modify another user's app bytecode or prevent them from installing apps with package names identical to ones installed by the attacker.* The shared package management mechanism that led to Hypothesis 4 is also the cause of other problems. Since package installation is platform-centric rather than user-centric, changes by any user authorized to install apps will affect all users on the platform. Specifically, if a secondary user upgrades a package, the bytecode changes affect all users that have that package installed. Likewise, if a malicious user installs a fake app with a real app's package name, all users are prevented from installing the real app. Results from testing this hypothesis are contained in Section 3.4.3.

To test these hypotheses, experiments were designed and conducted using builds of Android 4.4.2_r1 branch of the open source project [38]. The details of these experiments and findings are the subject of Section 3.4.

3.4 Case Study Findings

3.4.1 Unprotected Activities

Hypothesis 1 states that secondary users may be able to bypass their restrictions by exploiting the unprotected public interfaces of system apps. To find out if this is true, our experiment must first identify the intended restrictions placed on a secondary user, and then compare them with the full set of exposed interfaces.

To understand the *intended* restrictions on secondary users, the device UI elements accessible to a secondary user were systematically mapped and compared with those of the owner. A privileged app where significant differences have been observed is *Settings*. *Settings* is important to consider from a security point of view because it is granted `SignatureOrSystem` permissions such as `WRITE_SECURE_SETTINGS`. This permission allows *Settings* to make changes to variables defined in *Settings*' protected nested classes `Settings.Global` and `Settings.Secure` via the settings provider [39]. With these special permissions, *Settings* is the means by which the owner can perform device management tasks, such as adding, removing & restricting users, changing mobile data/plan settings, changing locations settings, changing WiFi settings, performing backups, etc.

From the UI mapping, we observed that *Settings* implements a number of UI restrictions based on type of user by hiding certain menu items. As such, we infer that these are capabilities that secondary users are not supposed to have. As an example, virtual private network (VPN) settings are hidden from the secondary user by way of logic within `WirelessSettings.java`, as shown in Listing 3.7. This logic compares the current user's *userId* with that of the owner and executes `removePreference(KEY_VPN_SETTINGS)` if not equal.

Listing 3.7: Settings code which hides VPN menus for secondary users.

```
public void onCreate(Bundle savedInstanceState) {
  ...
  final boolean isSecondaryUser = UserHandle.myUserId() != UserHandle.USER_OWNER;
  if (isSecondaryUser) { // Disable VPN
            removePreference(KEY_VPN_SETTINGS);
  }
  ...
}
```

With an understanding of how *Settings* presents a restricted UI to secondary users, we compared the list of restricted UI elements with exported activities in the app's manifest to find which of these elements can be launched directly via Intent [40]. Listing 3.9 contains an excerpt from *Settings*' `AndroidManifest.xml` showing an Intent filter in `VpnSettingsActivity`, implying that it can be launched from components of other applications [40]. As expected the code shown in Listing 3.8 was used in a test app to confirm that a secondary user could bypass the UI restrictions and access **VpnSettingsActivity** directly.

Listing 3.8: Code for direct access to ***VpnSettingsActivity***

```
Intent intent = new Intent();
intent.setClassName("com.android.settings",
        "com.android.settings.Settings$VpnSettingsActivity");
startActivity(intent);
```

Similar examples were found in Mobile network & Mobile plan settings (under Wireless & Network settings), and in Backup & reset settings (under Personal settings). Secondary users can access these activities because of a lack of access control along path ② of Figure 3.3a, such as a check based on **UserHandle.myUserId()**. Thus, each of these examples represent potentially dangerous situations since these activities allow a secondary user to manipulate configuration settings that may be able to be used to negatively affect the owner or other users of the device.

As it turns out, the VPN example contains additional access control checks in **Vpn.java** that do properly identify the subject and prevent restricted users from connecting VPNs. Thus, for VPNs at least, the hypothesis is only partially true. Because of the numerous cases of restricted UI elements also being exported to all users, other cases will be investigated in the future.

Listing 3.9: ***AndroidManifest.xml*** *excerpt showing an Intent filter in* ***VpnSettingsActivity****.*

```
...
<activity android:name="Settings$VpnSettingsActivity"
  ... >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <action android:name="android.net.vpn.SETTINGS" />
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.VOICE_LAUNCH"/>
    <category android:name="com.android.settings.SHORTCUT"/>
  </intent-filter>

  <meta-data android:name="com.android.settings.FRAGMENT_CLASS"
   android:value="com.android.settings.vpn2.VpnSettings"/>
  ...
  <meta-data android:name="com.android.settings.PARENT_FRAGMENT_CLASS"
   android:value="com.android.settings.Settings$WirelessSettingsActivity"/>
</activity>
...
```

3.4.2 Unrestricted Administrative Functions

Hypothesis 2 states that secondary users may be able to use device configurations which are persistent across user switches to attack other users. Although related to Hypothesis 1, this case does not involve a user bypassing restrictions, but simply implementing a malicious environment using the UI elements freely available to them.

To test this hypothesis, we built an experiment around network configuration, since this function is usually reserved for administrative users on standard multi-user platforms. We found that all users, secondary, restricted profile or otherwise, have full access to WiFi settings and can add and configure network connections as they choose. Furthermore, these settings are common to all users since they are ultimately stored by the system in a single **`/data/misc/wifi/wpa_supplicant.conf`** file that has no provisions for identifying the user who has authorized a particular connection. Finally, our experiment showed that WiFi connections are persistent across user switching.

This arrangement enables a secondary user to connect a multi-user device to a malicious hotspot and control all traffic to/from the device while it is being operated by other users. The hypothesis is true and the situation represents the fifth case mentioned in the Figure 3.3 discussion, that of *no* access control.

3.4.3 Shared Package Information

Hypotheses 4 and 5 state that specific problems may occur due to the fact that Android apps belonging to different users share critical package information:

1. Apps sharing the same **`appId`** in different users share permissions. As a result, the effective permission of these apps is the union of the declared permissions for each app and the **`sharedUserId`** apps escalate their permissions. This is the essence of Hypothesis 4.

2. An app installed for different users shares the same app package information. Consequently, one user may trigger a package update to modify the app's manifest file or code without other users' consent. This is Hypothesis 5.

To design an experiment to confirm these two problems, we need to first understand more about how *PackageManager* stores and uses an installed app's package and its relevant information. In *PackageManager*, all the package information among users are stored in a global hash map **`mPackages`** as shown in Listing 3.10.

Listing 3.10: Global hash map storing package information.

```
    // Keys are String (package name), values are Package.
    final HashMap<String, PackageParser.Package> mPackages =
            new HashMap<String, PackageParser.Package>();
```

The keys of this hash map are package names, and the values are packages including permissions and code information of the packages. Hence, we realize that **`mPackages`** is app name-based, rather than user-based, confirming the platform-centric approach to package management that remains in Android, in spite of the addition of the multi-user framework. With this as a basis for our understanding, we can now discuss the testing of each of these hypotheses separately.

3.4.3.1 Permission leakage in **`sharedUserId`** *apps*

Android's **`sharedUserId`** feature allows apps signed with the same key to share permissions and data. Previous work in the single user environment has shown this convenience feature to have risks due to implicit capability leaks among apps [41]. Although **`sharedUserId`** app's data ends up being properly isolated in multi-user due to Linux's use of the **`uid`** (which accounts for both **`appId`** and **`userId`**), this is not the case with permissions. In fact, these capabilities are leaked across user boundaries, even if a particular user only has one of the **`sharedUserId`** apps installed. This occurs because of the platform-centric design of **`PackageManager`**.

During installation, permission sets are stored in **`packages.xml`**, while installation status is stored in separate `package-restrictions.xml` files for each user. For `sharedUserId` apps, permissions from each app are combined within the `<shared-user>` block in `packages.xml`, as explained previously in Section 3.2.3. During boot, *PackageManager* loads this permission list into the hash map `mPackage` in a way that makes it impossible to separate the individual permissions from each `sharedUserId` app in case a particular user does not have them all installed. As a result, when `sharedUserId` apps from the same developer are installed in varying combinations by different users on the same device, every single app gains the union of permissions from all of the `sharedUserId` apps installed on the platform, regardless of which `sharedUserId` apps have been installed by that particular user.

To confirm this, we created a pair of `sharedUserId` apps. *shareduidapp1* declares `INTERNET` permission, and *shareduidapp2* declares `READ_CONTACTS` permission. We then installed *shareduidapp1* under the owner's account only, and *shareduidapp2* under a secondary account only. After these installations, `packages.xml` contained the entries shown in Listing 3.11:

Listing 3.11: `packages.xml` excerpt showing permissions associated with shareduidapp1 and shareduidapp2.

```
<package name="com.example.shareduidapp1" ... sharedUserId="10056">
<package name="com.example.shareduidapp2" ... sharedUserId="10056">
<shared-user name="com.example" userId="10056">
    <perms>
        <item name="android.permission.READ_CONTACTS" />
        <item name="android.permission.INTERNET" />
    </perms>
```

As Listing 3.11 shows, *shareduidapp1* and *shareduidapp2* share `userId` 10056 per the `<package>` blocks. Separate from the package names, within the `<shared-user>` block, `userId` 10056 is then associated with the two permissions. However, no structure retains the fact that the `INTERNET` permission was contributed by *shareduidapp1*, while `READ_CONTACTS` was contributed by *shareduidapp2*. The record of which users have these apps installed is contained in each user's separate `package-restrictions.xml` file. As Listings 3.12 and 3.13 show, user 0's (Owner) `package-restrictions.xml` shows `inst=false` for `shareduidapp2`, while user 10's (secondary user) shows `inst=false` for `shareduidapp1`.

Listing 3.12: ***`package-restrictions.xml`*** *for user 0 (Owner).*

```
...
<pkg name="com.example.shareduidapp2" inst="false" stopped="true" nl="true" />
...
```

Listing 3.13: ***`package-restrictions.xml`*** *for user 10 (secondary user).*

```
...
<pkg name="com.example.shareduidapp1" inst="false" stopped="true" nl="true" />
...
```

Because of the commingling of permissions within `packages.xml`, when user 10 runs `shareduidapp2`, the system grants both `INTERNET` and `READ_CONTACTS` permissions even though `shareduidapp1` is not installed for the user. Meanwhile, *Settings* reports `shareduidapp2` only holds `READ_CONTACTS` permission. This condition also occurs for `shareduidapp1` run by user 0. Each user is unaware of the permission leakage and over-privilege. Moreover, if user 10 is a restricted profile for which the owner carefully enabled apps based on their reported permissions, this leakage could allow the restricted profile accesses they should not have.

3.4.3.2 Package-based code sharing across users

Android's app install and update procedure is depicted in Figure 3.4. When *PackageManager* receives an install request, it first checks whether the package has been previously installed on the platform. If the package has been previously installed by at least one user, it's treated as a package replacement. Otherwise it's treated as a new install.

For new installs, a new mapping is created in the hashmap `mPackages`, and the app is marked as installed for the installing user (or in some cases, all users) by passing the appropriate *userId*(s) to the `setInstalled` method of the new package's `PackageSetting`, as shown in Listing 3.14. Hence, *package meta-data*, in the form of an instance of `PackageSetting`, solely determines

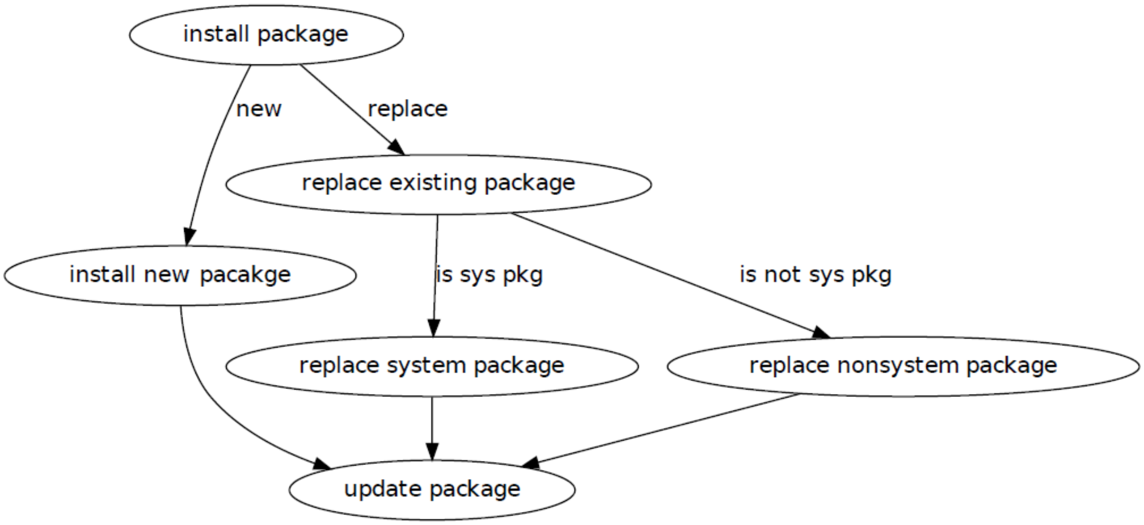


Figure 3.4: Package installation and update procedure.

whether the package is installed for a particular user. This is further evidenced by the code of method `installExistingPackageAsUser` within *PackageManagerService*, shown in Listing 3.15. Here we see that an existing package is installed for a particular *userId* simply by changing the values within an instance of `PackageSetting`.

Listing 3.14: Per-user package installation state is maintained within package meta-data.

```
PackageSetting p = mPackages.get(name);
...
        // The caller has explicitly specified the user they want this
        // package installed for, and the package already exists.
        // Make sure it conforms to the new request.
        List<UserInfo> users = getAllUsers();
        if (users != null) {
            for (UserInfo user : users) {
                if (installUser.getIdentifier() == UserHandle.USER_ALL
                        || installUser.getIdentifier() == user.id) {
                    boolean installed = p.getInstalled(user.id);
                    if (!installed) {
                       p.setInstalled(true, user.id);
                       writePackageRestrictionsLPr(user.id);
                    }
                }
            }
        }
```

Listing 3.15: Existing packages are "installed" for additional users by modifying the package meta-data.

```
public int installExistingPackageAsUser(String packageName, int userId) {
    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.INSTALL_PACKAGES,
            null);
    PackageSetting pkgSetting;
    ...
            if (!pkgSetting.getInstalled(userId)) {
                pkgSetting.setInstalled(true, userId);
                pkgSetting.setBlocked(false, userId);
                mSettings.writePackageRestrictionsLPr(userId);
                sendAdded = true;
            }
    ...
```

Since installation state for each user is maintained within the package meta-data, it is impossible for users to maintain different versions of the same package. In fact, *PackageManager* uses the package signature to ensure that updates *replace* old versions of the package. Listing 3.16 shows that a package update passing the signature check is then used to replace the old package by way of `replaceSystemPackageLI` or `replaceNonSystemPackageLI`. Moreover, per-user package installation state is preserved with the new package installation, as evidenced by the passing of `perUserInstalled` to these methods. As a result of this design, a user who updates a package updates it for all other users of the device.

Listing 3.16: Package signature is checked prior to replacement during an update.

```
 private void replacePackageLI(PackageParser.Package pkg,
            int parseFlags, int scanMode, UserHandle user,
            String installerPackageName, PackageInstalledInfo res) {
     ...
     oldPackage = mPackages.get(pkgName);
     if (compareSignatures(oldPackage.mSignatures, pkg.mSignatures)
             != PackageManager.SIGNATURE_MATCH) {
         Slog.w(TAG, "New package has a different signature: " + pkgName);
         res.returnCode = PackageManager.INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES;
       return;
     }
     ...
   boolean sysPkg = (isSystemApp(oldPackage));
   if (sysPkg) {
       replaceSystemPackageLI(oldPackage, pkg, parseFlags, scanMode,
               user, allUsers, perUserInstalled, installerPackageName, res);
   } else {
       replaceNonSystemPackageLI(oldPackage, pkg, parseFlags, scanMode,
               user, allUsers, perUserInstalled, installerPackageName, res);
   }
 }
```

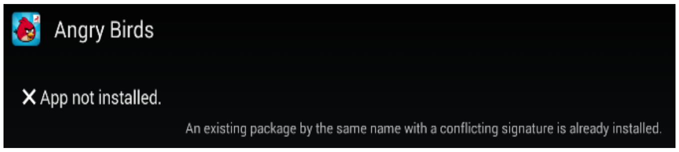
The most significant security impacts of this design are as follows:

First, one user may escalate the permissions of apps belonging to a second user. For example, the latest version of Twitter requires an extra permission, READ_SMS, compared to the old version. The owner may choose not to upgrade to the latest one for privacy concerns related to SMS. However, a secondary user may choose to update the app through Google Play because she likes the new features. As a result, this update event will update all users' version of Twitter without their consent. The newly updated package requests more permissions and performs different computing logic than the old one. In this scenario, a secondary user grants a new permission to Twitter on behalf of all the users instead of just herself.

Second, a user may have a chance to affect other users' app installation by creating denial of service (DoS) attacks in two ways. First, a user can fake a package and create package installation denial-of-service (DoS) by installing a fake version of an app before other users install the legitimate one. In such a case, because of the signature matching requirement, no one else can install the legitimate app, or uninstall the faked package through the user interface. Only the owner can force uninstalls using adb. Moreover, if other users do not notice that the app is fake, the attacking user can update the fake app and include malicious logic that attacks other users' sensitive information.

Another negative side-effect of all users sharing the same appId, is that one user may use up all the appId values which prevents other users from installing any apps. We confirmed this by installing 50,000 dummy apps on a Nexus 10 running KitKat 4.4 as a secondary user, thus using up all available appIds. As a result, any other user, including the owner, cannot install apps anymore. The logcat will show that the installation failure is because INSTALL_FAILED_INSUFFICIENT_STORAGE, but actually there is still space in data partition. The failure is because all users must share the same appId range.

The root cause for this code sharing problem is that Android does not provide code separation for different users. All users share the same code for each package, appIds, and their privileges for installing apps is mixed together. The package manager fails to isolate the code space of each



Angry Birds

✕ App not installed.

An existing package by the same name with a conflicting signature is already installed.

Figure 3.5: New package installation is denied due to existing package with the same name but different signature.

user, although this design significantly saves the valuable disk space.

3.4.4 Use of Sensors and Hardware Devices by Multiple Users

Hypothesis 3 from Section 3.3.3 was tested by a colleague, and therefore only summarized here as a way to provide additional evidence of how the methodology described earlier was put to use. Further details pertaining to the sensor and hardware testing are found in [34].

In this section, we aim to answer Hypothesis 3 from Section 3.3.3. That is, non-logged in secondary users can exploit improper access control enforcement on shared hardware resources to spy on current users. In fact, if Android does not enforce proper access control on shared hardware resources based on user status, a non-current user can still use a hardware interface to infer various information about the logged in users and spy on them. For example, if a non-current user can query the light and accelerometer sensors over a time interval, he can infer potential activities about current user such as whether he is sitting indoors, or jogging outdoors. Moreover, if he can query the GPS service, he can even infer where he is sitting or jogging. Even more concerning, if he can launch the sound and camera recorders, he can know easily more details such as with whom he is and what type of conversation he is having.

Under single-user assumptions, all hardware interfaces belong to the same user without any concerns of misuse. Ideally, with multi-user features, a hardware resource should only be bound to a single user at a time, corresponding to the currently logged-in user. Since the hardware interfaces are shared among the users on an Android device, the transition from single to multiple user framework requires changing the access control model on all hardware resources to make sure that use of a hardware resource is only granted to the logged in user.

To ensure that a hardware resource is only bound to the currently logged in user, Android should be able to identify if the user requesting a resource is logged in. Also, it should track if the user who initiated the request is continuously logged in during the service lifetime. More specifically, if user-switching occurs, Android should be able to revoke any resource access from non-logged in users. Thus, the hypothesis testing in this section focuses on whether or not Android access control enforcement for shared hardware resources factors in user status. Media resources and common sensors are each tested against the hypothesis: for each resource, an test app is designed that will attempt to access a resource even if the user running the app is not logged in. This approach exploits the fact that *ActivityManager* does not kill all non-current user processes. Thus, the attacking app can be launched when the malicious user is logged in, and continues to run after he logs out and the victim user logs in. A non-owner user is deliberately chosen to be the attacker, and the owner to be the victim since non-owner is less privileged compared to the owner and represents the worst case. Findings for each of the resource categories are summarized below.

3.4.4.1 Media resources

To check if relevant access control points take into account multiple users, an test app was designed to launch the camera (without a preview window) and start video recording while the victim is using the device. The app is launched from the attacker user account (*userId* 10) and then the device is switched to the victim account (*userId* 0). The test app was observed to continue recording video while the victim is using the device. Since the test app is running as the attacker, the recorded video is saved under the attacker's data directory and can be retrieved by the attacker later. The success of the attack reveals that the media resource access controls only come into play at request-time and do not consider the changing of user status.

3.4.4.2 Motion, environmental and position sensors

Most Android devices have numerous built-in sensors such as motion, environmental, and position. Motion sensors include accelerometers, gravity, rotational vector sensors, and gyroscopes. Environmental sensors measure ambient air temperature, pressure, illumination and humidity, while position sensors measure the physical orientation of a device. Unlike the media devices discussed above, activity from these sensors follow an event-driven approach. In other words, an app first registers a listener to receive sensor events through the `SensorManager`, then *SensorService* will deliver the sensor data to the registered listeners.

To test whether Android's sensor access controls consider user state, a test app was developed to continuously log sensor data. Similar to how the video recording app was used, the sensor app is launched from the attacker account and then the device is switched to the victim user. Even after user switching, the app continues to receive sensor events, silently while the victim is using the device. Sensor data logs are stored in the attacker's data directory and may be retrieved later. This success indicates that no access controls exist for the sensor devices, either at listener registration time or during sensor data delivery. The conclusion is that the sensor subsystem fails to consider the new

3.4.4.3 Location sensor

Tests similar to those above were performed on the GPS location sensor and found that a non-current user cannot succeed in getting GPS location updates of the logged in user. A review of *LocationManagerService* code revealed that it does indeed apply proper access controls that consider user status when location data is dispatched. Specifically, **`handleLocationChangedLocked()`** will only dispatch location updates to registered listeners belonging to the current user. In this case, *LocationManagerService* properly tracks the current user and updates its instance variable, **`mCurrentUserId`**, each time the user is switched. A similar design should be applied to other shared hardware resources.

Chapter 4

Access Control Characterization

You can't secure what you don't understand.

- Bruce Schneier, *Schneier on Security, 1999* [42]

4.1 Introduction

The previous chapter highlights the need to consider Android's unique factors when evaluating or improving the platform's security. These factors include its open design, decentralized, resource-centric access controls, and emphasis on usability. Besides these, the phenomenon of vendor and carrier customization is another reality that sets Android apart from other mobile operating systems—and introduces a whole new dimension of security concerns.

Nearly all of the Android devices in use throughout the world are those which have been customized by manufacturers and service carriers. Unlike devices running the open-source AOSP baseline, these customized devices run versions of Android that are proprietary and closed source. Designed to create a competitive advantage through product differentiation, typical modifications include the addition of pre-loaded apps, custom launchers, mobile device management (MDM) features, carrier-specific enhancements or restrictions. Previous work has studied the security implications of some of these modifications, including hanging attributes references [43], problems with pre-loaded apps [44,41,45], security configuration changes [46], and access control inconsistencies [47]. Other work has focused on automated detection of bugs in the Linux kernel [48,49,50] and and inconsistencies in its configuration options [51,52]. These and other related works are discussed in more detail in Chapter 6.

4.1.1 System Services Customization

One type of customization that has not been thoroughly explored, but is seen in nearly all vendor images, is that of customized System Services. As described in Sections 2.2.4 and 5.4.1, Android employs a modular System Services architecture, whereby system resources are accessed via separate managers, such as *LocationManager* and *TelephonyManager*. These managers communicate via IPC with the corresponding service which in turn accesses the actual hardware or software resource. The service is responsible for enforcing its own access control policy. Under this resource abstraction, specific resources can be presented via one or more managers with different APIs. For example, *NetworkManager* and *ConnectivityManager* both manipulate the device's networking infrastructure, but via different APIs intended for different purposes. This modularity allows 3rd parties to easily add and modify System Services, creating custom resource interfaces without affecting other parts of the system. Unfortunately, since access control is implemented in each service rather than centrally, it also opens the door for new vulnerabilities and access control inconsistencies if not done properly.

4.1.2 Motivating Example

To illustrate the types of customization problems that can occur, consider the following example. In Lollipop 5.0, AOSP's *NetworkManagementService* contains 81 remotely-callable public interface methods which collectively are protected by 64 permission enforcement points that check the

CONNECTIVITY_INTERNAL permission. Since this is a *signatureOrSystem*-level permission, it is only obtainable by apps signed with the platform key or residing in the system partition (i.e., trusted system apps). In *no* instances does AOSP's *NetworkManagementService* rely on credentials obtainable by 3rd-party apps (i.e., *normal*- or *dangerous*-level permissions) to enforce access control. As such, it is evident that Android's designers intended this service to be used exclusively by system apps.

In contrast, Motorola's customized *NetworkManagementService* obtained from a Moto X device running their version of Android 5.0 contains 8 additional remotely-callable methods: `addUpstreamV6Interface()`, `blockDataTrafficInternal()`, `enableTrafficMonitor()`, `getSapAutoChannelSelection()`, `getSapOperatingChannel()`, `removeUpstreamV6Interface()`, `runIpLogCmd()`, and `setChannelRange()`. None of the other 81 methods are changed from their AOSP version.

Of the eight added methods, only two, `enableTrafficMonitor()` and `blockDataTrafficInternal()`, require system-only credentials such as those in AOSP's *NetworkManagementService*. Specifically, `enableTrafficMonitor()` requires *CONNECTIVITY_INTERNAL* permission, while `blockDataTrafficInternal()` requires the calling process to be running as *uid* 1000 (a privileged *uid*, reserved for system processes).

Those that appear to deviate from Google's system-only design for *NetworkManagementService* include `addUpstreamV6Interface()` and `removeUpstreamV6Interface()`, which are protected with the *normal*-level permission *ACCESS_NETWORK_STATE*. Also, `getSapAutoChannelSelection()`, `getSapOperatingChannel()`, and `setChannelRange()` are protected with the *dangerous* permission *CHANGE_WIFI_STATE*. Finally, `runIpLogCmd()` has no apparent access controls at all. A test app was written to verify that these six methods could indeed be invoked by a non-system app with only *normal*- or *dangerous*-level permissions. Each was successfully invoked.

4.1.3 Research Questions

This example illustrates several aspects of system customization that are important to understand, and thus translate into the research questions for this investigation. First, *“what is the type and nature of access controls in System Services?”* Second, *“has a vendor customized System Services for their devices and, if so, how?”* Finally, *“if present, how do the customizations compare with a known baseline, AOSP?”* Answering the first research question requires a *characterization* of access control, while answering the second two requires a *comparison*, or *differential analysis*, of the vendor image vs. the baseline AOSP. As such, this chapter first introduces a new method for characterizing Android access controls (Section 4.2), and then describes how these characterizations are compared to highlight interesting aspects of their differences (Section 4.3).

4.2 Characterizing Android Access Controls

The purpose of any characterization is to describe the distinctive features of something so that it can be distinguished from otherwise similar objects. For example, accurate automatic facial recognition relies on careful selection and quantification of various facial features extracted from an image of the subject. These features alone are then used in distinguishing one subject from another or when matching against a library of subjects. Thus, to accurately characterize access controls of a particular entity in Android, we must first determine the features that are most useful for capturing the nature of the entity’s access controls and in turn allow it to be distinguished from its customized counterpart. Before proposing a feature set, we capture some specifics of Android access control and its implementation. To this end, we begin with “first principles” and consider the high-level diagram of Figure 4.1, as well as the background material provided in Chapter 2.

The core of this diagram shows a *subject*, $s \in \mathit{Subj}$, requesting access to an *object*, $o \in \mathit{Obj}$, by way of a central *access control mechanism*, $m \in \mathit{Mech}$. This well-known simple model is extended in two ways here. First, an *assignment* step (①) is added to indicate how the subject gets its

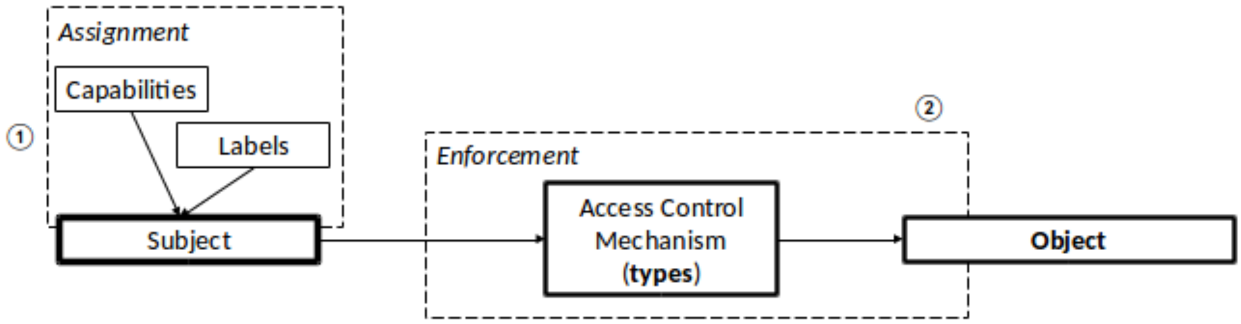


Figure 4.1: High-level access control model.

privileges when it's created or elevated. These privileges, $p \in Priv$, come in the form of capabilities that the subject possesses (such as a Binder token), or labels (such as *uid* or *gid*). Second, an *enforcement* boundary (②) is shown to indicate that the set of mechanisms may include those that are implemented by the resource, as well as those that are central system mechanisms.

In order to make this generic diagram useful for representing Android access controls, the following questions must be answered for Android:

1. What are the possible subjects, *Subj*?
2. What labels and capabilities, $Priv$, can be assigned to the subject?
3. What enforcement mechanisms, *Mech*, are used for access control?
4. What are the possible objects, *Obj*?
5. Are there other interesting access control-related mechanisms?

In general, there are several ways in which these questions could be answered, including comprehensive code analysis, by inferences made from system observations, from documentation, or information provided by experts. As a starting point, since formal specifications for Android do not exist, a collection of security documents from the Android Open Source Project (AOSP) [10,7] and several key papers [20,53,37] were manually parsed for security-related assertions. This analysis found 146 security statements, 110 of which were related to access control (see Appendix A). From these, all mentions of different subjects, objects, assignments, and enforcement mechanisms were extracted. These lists were further refined and confirmed using the results of the subject-object-path identification process described in Chapter 3. The result, summarized below, provides answers for the questions above:

1. What are the possible subjects? Subjects are initiators of requests governed by access controls. In Android, these are apps, services and native processes. According to the default SE for Android [53] policy, apps are further refined as isolated apps, platform apps, system apps, untrusted apps, and shell apps. This refinement is important in this model since the assignment process is different for different types of apps. For example, platform and system apps can obtain special permissions that untrusted apps cannot. Thus,

$$Subj = \{isolated\_app, platform\_app, system\_app, untrusted\_app, service, native\} .$$

2. What labels and capabilities can be assigned to the subject? Assignment is when the subject get its powers. These powers come in the form of labels and capabilities. Labels are simply properties or meta-data that are used to identify the subject during access control decisions. In Android, labels include Linux user ID (*uid*), Linux group ID (*gid*), SELinux security identifier (*SID*), **`appId`**/**`userId`** (these are related to **`uid`** [34]), package name, and signing certificate.

Manifest permissions are assigned to apps at installation and run-time. Permissions have a *protection level* of *normal*, *dangerous*, *signature*, or *signatureOrSystem*, corresponding to the potential risk involved, and affecting which subjects can obtain them and whether the user is consulted when they are granted [54].

Capabilities are tokens (tickets) that are actually held by the subject and presented to the enforcement system when access is needed. In Android, Binder tokens and file descriptors are examples of capabilities that are used to control access. Hence,

$$Priv = \{uid, gid, SID, package\_name, cert, perms, Binder\_token, file\_desc\} ,$$

where *uid* is a combination of `userId` and `appId`, and *perms* is the set of all possible manifest permissions.

A key insight that was gained during this analysis is related to label assignment. It was found that assignments may take two forms, which this work terms *dynamic* and *fixed*. Dynamic assignments are those that are dependent on some external input or environment. For example, the device owner may choose to deny certain permissions when installing an app or make the app available to only certain secondary users. In another example, assignment of *signatureOrSystem* permissions is dependent on whether the app is signed with the platform key or located within the system partition. Apps not meeting one of these criteria will be denied the system permission. In both of these cases, the resulting label(s) depend on the specific circumstances at the time of assignment. On the other hand, fixed assignments are those that are hardcoded into the system, such as the permissions assigned to native daemons by `init` per `init.rc`, or the SID labels applied to files and directories when the system image is built. Thus, the assignment itself may be subject to access controls, and some of these are *discretionary* (i.e., up to the user) and some are mandatory (i.e., hardcoded or fixed by mandatory policy).

3. What are the enforcement mechanisms used for access control? When a subject requests something, it is usually the system's job to make an access control decision based on the subject's identity, capabilities it may posses, and system policy. In many cases in Android, access control points are implemented by resources themselves and may factor into the chain of access controls. The study identified five basic types of enforcement mechanisms in Android:

- *Capability-based*. In the Binder driver, these come in the form of Binder tokens which are issued to processes requesting access to IPC targets like services and other apps. The kernel Binder driver maintains a structure of issued tokens for each process so as to enforce this access control. This category also includes standard Linux capabilities such as file descriptors which are also enforced by the kernel.
- *Linux pseudo-capability-based*. When apps are launched as new processes forked by zygote,

various Linux capabilities are added or dropped from the forked process. While not true capabilities, these nonetheless represent specific powers that the forked process may possess. These are enforced by the Linux kernel in the same way as in traditional Linux systems.

- *Linux id-based*. Every file and running process is owned by a Linux *uid*, one or more *gid*s, and is labeled with an SELinux *SID*. Running processes are also assigned a Linux process identification (*pid*) and SID. All of these identities are maintained and protected by the kernel and are used by traditional Linux access control mechanisms and, with the exception of SID, relied upon throughout the Android Framework. In the Framework, these checks are based on the caller's id as reported by the kernel Binder driver that is mediating the IPC.
- *Android permissions-based*. At installation-time, a record of an app's permissions, as requested via the manifest file, is stored by *PackageManager*. When an app makes a request, the presence of a needed permissions is looked up by the system and/or the resource requested. This mechanism is exclusive to the Framework, as manifest permissions have no meaning to the kernel. Enforcement points typically include calls to `checkPermission()` or its variants.
- *User-based*. As explained in Section 3.3.1, some access control decisions are accomplished via direct interaction with the user. Many of these involve actions that would incur financial obligations, such as premium SMS, or revealing private information, including location, contacts, and photos. With the recent addition of run-time permissions to Android, the role of this mechanism has been greatly expanded.

In summary,

$$Mech = \{cap, linux\_cap, id, perm, user\} .$$

Important to the discussion of access control mechanisms is the fact that ultimate access to a particular Android resource usually requires several steps and involves more than one of the above mechanisms. For example, access to a resource managed by a system service requires the app to first obtain a capability (Binder token) for the service, which is permitted or denied based on the app's *uid* and *SID*. Once the app holds the capability, it may attempt to communicate with the service via IPC, which is allowed only if the caller continues to hold the capability and the kernel can verify this. Finally, the app's request must pass any access controls present in the the service method being remotely called. These are usually permission- or Linux id-based, and may also include user-based confirmations.

4. What are the possible objects? Objects are the target of an access control request. The document analysis and enumeration of Section 3.3 reveals that objects in Android include System Services, apps (when their components are called by others), Linux files and sockets, and many different Framework objects including the certificate store, modular frameworks such as Device Administration, *DreamService*, etc.

5. Are there other interesting access control-related mechanisms? Security decisions in Android are based on the identity of the subject, $Subj$. However, there are times when System Services, acting on behalf of the subject, need to perform operations that the subject does not have permission for. For example, before returning the appropriate *Location* object, `getLastLocation()` in *LocationManagerService* first checks the location blacklist, whether the available location provider(s) are allowed, how often the app has requested location, and if the location object contains a mask flag. These internal operations require privileges not granted to apps, so they are placed in a block wrapped with *clearCallingIdentity()* and *restoreCallingIdentity()*. *clearCallingIdentity()* resets the identity on the current thread so that these operations are peformed with the privileges of the `systemserver` process, not the caller. When finished, *restoreCallingIdentity()* restores the identity on the current thread to that of the orignal caller.

4.2.1 Access Control Feature Set

Knowing the sets of possible subjects, objects, labels and access control mechanisms in Android allows for the definition of a *feature set* that can capture the nature of access control present in code. Just as in facial recognition algorithms that rely on extraction of geometric feature sets, the access control feature set does not attempt to capture *every* aspect of the object, but strives to represent certain aspects well enough to be useful. Moreover, the object's representation in terms of a feature set should be resilient to uninteresting changes in the object. For example, effective facial recognition systems need to be robust against superficial changes such as skin tone, scars, or facial hair. While these may change the original image enough that a direct comparison would fail, a set of numerical features that capture the unique geometry of the face can be more efficient as well as independent of the superficial changes. In the same way, the access control feature set should be independent of superficial code changes, such as the addition or removal of logging statements.

Choosing the right set of features to accomplish this is non-trivial and may require significant domain knowledge. This process of feature definition is known as *feature engineering*. Feature engineering uses domain knowledge to identify numerical features for use in machine automation [55]. In feature engineering, intuition and domain-specific insights into what's being represented are just as important as the technical aspects of using the features for automation [56]. The insights and experience gained from the research and solutions described in the previous chapters, combined with the initial *NetworkManagementService* study and systematic Android access control analysis described above, form the foundational domain knowledge necessary to establish a useful feature set.

The focus of this work, Android System Services, are typically large Java classes containing public, private and protected methods. In addition, a subset of each service's public methods are remotely callable via Android's Binder IPC. These are referred to here as *AIDL* methods, and are special interest since they represent the entry points accessible to untrusted apps and potential malware. Hence, it is of first-order importance that the AIDL methods be fully understood in terms of access control. As such, the Android access control feature set, *Feat*, used in this work is as follows:

$$\begin{aligned}
Feat = \{&methodType, isAIDL, getCallingUid, getCallingPid,\\
&clearCallingIdentity, restoreCallingIdentity, checkPermission,\\
&checkCallingOrSelfPermission, checkCallingPermission,\\
&enforcePermission, enforceCallingPermission,\\
&enforceCallingOrSelfPermission, securityException,\\
&permissionNormal, permissionDangerous, permissionSig,\\
&permissionUndef\},
\end{aligned} \tag{4.1}$$

where

$methodType$ = *public*, *private*, or *protected*,

$isAIDL = 1$ if method is remotely callable; 0 otherwise,

$getCallingUid = 1$ if method calls `getCallingUid()`; 0 otherwise,

$getCallingPid = 1$ if method calls `getCallingPid()`; 0 otherwise,

$clearCallingIdentity = 1$ if method calls `clearCallingIdentity()`; 0 otherwise,

$restoreCallingIdentity = 1$ if method calls `restoreCallingIdentity()`; 0 otherwise,

$checkPermission = 1$ if method calls `checkPermission()`; 0 otherwise,

$checkCallingOrSelfPermission = 1$ if method calls `checkCallingOrSelfPermission()`; 0 otherwise,

$checkCallingPermission = 1$ if method calls `checkCallingPermission()`; 0 otherwise,

$enforcePermission = 1$ if method calls `enforcePermission()`; 0 otherwise,

$enforceCallingPermission = 1$ if method calls `enforceCallingPermission()`; 0

otherwise,

enforceCallingOrSelfPermission = 1 if method calls `enforceCallingOrSelfPermission()`; 0 otherwise,

securityException = 1 if method can raise a *SecurityException*; 0 otherwise,

permissionNormal = 1 if method is protected with a *normal*-level permission; 0 otherwise,

permissionDangerous = 1 if method is protected with a *dangerous*-level permission; 0 otherwise,

permissionSig = 1 if the method is protected with a *signatureOrSystem*-level permission; 0 otherwise, and

permissionUndef = 1 if the method is protected with a unknown-level permission; 0 otherwise.

For every method m in the class under study, a *feature vector*, f_m is extracted that contains specific values corresponding to each element of Feat. By way of example, Listing 4.1 shows the Java source code for method `addUpstreamV6Interface()`, reconstructed from the `.class` file containing Motorola's customized *NetworkManagementService* discussed above.

Listing 4.1: Reconstructed Java source for method `addUpstreamV6Interface()` in Motorola's customized NetworkManagementService.

```java
public void addUpstreamV6Interface(String paramString)
  throws IllegalStateException
{
  this.mContext.enforceCallingOrSelfPermission("android.permission.ACCESS_NETWORK_STATE", "
      NetworkManagementService");
  Slog.d("NetworkManagementService", "addUpstreamInterface(" + paramString + ")");
  try
  {
    NativeDaemonConnector.Command localCommand = new NativeDaemonConnector.Command("tether",
        new Object[] { "interface", "add_upstream" });
    localCommand.appendArg(paramString);
    this.mConnector.execute(localCommand);
    return;
  }
  catch (NativeDaemonConnectorException paramString)
  {
    throw new IllegalStateException("Cannot add upstream interface");
  }
}
```

A review of this code for the features defined in Feat reveals that $\mathit{methodType} = \mathit{public}$ and $\mathit{enforceCallingOrSelfPermission} = 1$. Also, since the permission *ACCESS_NETWORK_STATE* is defined as a *normal*-level permission in the image's manifest, $\mathit{permissionNormal} = 1$. Finally, by matching the method to one defined by the corresponding interface, *INetworkManagementService*, $\mathit{isAIDL} = 1$. Therefore, the feature vector for this method is as follows:

$$f_{\mathit{addUpstreamV6Interface}} = [\mathit{public}, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0]$$

.

4.2.2 Feature Vector Extraction

In production Android devices, there can be dozens of System Services containing thousands of methods and representing 10s of thousands of lines of code. Practical extraction of the feature vectors for every method in every System Service therefore requires automation. For the feature set defined above, standard static analysis tools can easily perform the task of analyzing code and writing feature vectors to a database. The feature vectors used for the results presented herein were automated using Java static analysis to extract feature vectors from key `.jar` files found in Android system images. An overview of the entire extraction process is shown in Figure 4.2 and summarized below.

The process begins with either an actual device or image archive file. If an actual device is available, and it can be rooted, a shell on the device itself can be used to extract the system partition (①) by `dd`ing the partition to a file, `system.img`. This dumped image can then be transferred off the device and mounted via the Linux *loop* device which enables access to the files within the system partition.

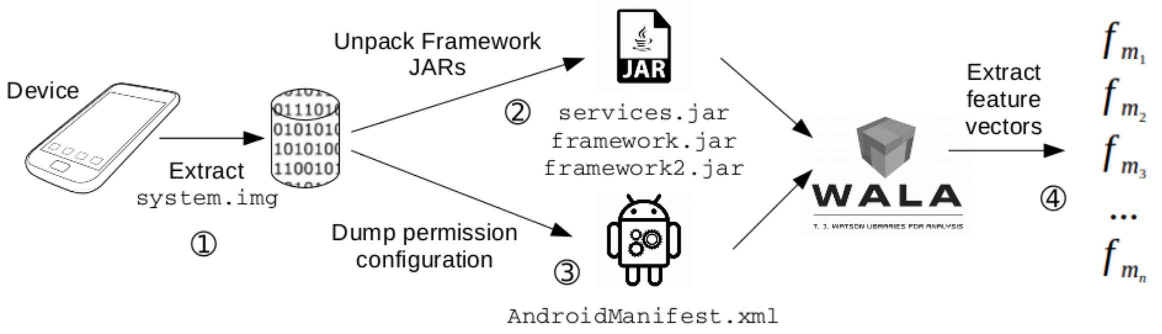


Figure 4.2: Process for extracting access control features from an Android device.

Image archive files, typically available from vendor support sites, can be unpacked and their contents accessed using procedures similar to those described in Appendix B for the images studied in this work.

In either case, of interest to the feature extraction described here are the JAR files **`services.jar`**, **`framework.jar`** and, if present, **`framework2.jar`** (②). **`services.jar`** contains most of the service manager classes instantiated by apps (e.g., *LocationManagerService*), while **`framework.jar`** and **`framework2.jar`** contain the interface classes containing the stub and proxy subclass implementations of the interface (e.g., *ILocationManager*).

`AndroidManifest.xml` files (③) from each app installed in the image are also required to determine the actual configuration of permissions on the device (i.e., permission protection levels and custom permissions). This information is obtained by using **`apktool`** to decode all of the APK files in the image, followed by parsing of the manifest files to find permission names and associated protection levels. Code for accomplishing this is shown in Appendix C.

Extraction (④) of the actual feature vectors specified here is accomplished with *FeatureExtraction*, a project developed by a colleague using the WALA libraries [57] to extract the features specified above. This code takes the JAR files and permission configuration file as input and writes a feature vector file for each System Service found in the image. The functionality of this code is summarized here.

From the WALA *ClassHierarchy*, System Services present in the JARs are discovered by searching the call graph for methods that register services to *ServiceManager* with `addService()` or `publishBinderService()`. The declaring class for methods using these are stored as a binder name and service name key-value pair in a hashmap, *SSClasses*.

Each class in *SSClasses* has a *ServiceClassRecord* that identifies the outer class, interface classes and inner classes for the service. All of the methods from each of these are combined to form a call graph which is then used for feature extraction. Each node in the graph is checked for the presence of a feature. If the feature is permission-related (e.g., *checkPermission()*, *enforceCallingOrSelfPermission*, etc.), the permission configuration data is consulted so that the permission-level features can be populated. Finally, a hash map is populated with the state of all features and then written to the feature vector file as comma-separated values. For `addUpstreamV6Interface()` discussed above, the record containing the feature vector is written to the file `NetworkManagementService.csv` as follows:

```
“[AIDL] public < Application, Lcom/android/server/NetworkManagementService,
addUpstreamV6Interface(Ljava/lang/String;)V >”,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0
```

Processing of the images studied here was accomplished on an 8-core i7 laptop with 32GB of main memory running WALA master branch[1] and Super CSV[2] in Eclipse Platform[3] v3.8.1. Generally, processing times for complete feature extraction ranged from less than an hour for some images to 10s of hours for others. Some highly-customized images, with many additional System Services required these longer processing times. For a small number of services, WALA threw exceptions and feature extraction for those services failed. This is believed to be related to errors encountered during deodexing (see Appendix B).

[1] https://github.com/wala/WALA

[2] http://super-csv.github.io/super-csv

[3] http://www.eclipse.org/

4.3 Comparing Android Access Controls

To highlight changes introduced by the vendor, a differential analysis of the feature vectors may be performed. This could be done during static analysis if both AOSP and vendor JARs are available simultaneously. Although this may shorten static analysis times, it would preclude full characterization of the image since only differences would be subject to feature extraction. Instead, the approach taken here is to difference the corresponding AOSP and vendor full System Service feature vector files generated earlier and store the result as a file that contains only feature vectors for added and modified methods. These files can then be analyzed to gain insights about the vendor modifications.

Differencing is accomplished by way of standard Linux scripts. First, each image's feature vector files are filtered for AIDL methods and combined into a single CSV for that image, `all_aidl.csv`. Next, pairs of these combined files are compared using `diff` in unified mode, as follows:

```
diff -u <baselineImageDir>/all_aidl.csv <vendorImageDir>/all_aidl.csv | grep ^+ >
<baselineImageDir>_<vendorImageDir>_AIDL_change.csv
```

Following this, some minor processing of the file is needed to remove unnecessary `diff` lines and add a column index header. The result is a single file which contains a method-level list of every feature vector from the vendor image that is different in some way from the corresponding AOSP feature vector.

4.4 Case Study

The utility of the access control feature vectors and differencing procedure introduced above is demonstrated by way of a case study on AOSP and vendor images in which the research questions above were addressed. Specifically, for characterization, "what is the type and nature of access controls in System Services?" For comparison, "has a vendor customized System Services for their devices and, if so, how?” and “if present, how do the customizations compare with a known baseline, AOSP?” Ultimately, these questions lead to the question of whether there are potential vulnerabilities introduced by the customization. This question will also be addressed in the case study.

Images listed in Table 4.1 were gathered from real devices, vendor support sites, and AOSP’s “stock” firmware site.

4.4.1 Procedure

First, each of the images listed in Table 4.1 was processed through the JAR deodex and feature vector extraction process. The resulting set of comma-separated values (CSV) files together contain over 1.8 million feature vectors for all methods in all System Services that were successfully decompiled and statically analyzed with *FeatureExtraction*. These CSVs were placed in a folder hierarchy with a common root node, and then imported into Microsoft®Excel®by way of the *Power Query Formula Language (PQFL)* script shown in Appendix D. In total, for all of the images and services analyzed combined, the resulting database contains 35,802 AIDL method feature vectors. Difference files are similarly imported using PQFL. In this case study, the database contains 8,037 feature vectors representing added or modified System Service methods.

4.4.2 Characterization Analysis

Once imported, a pivot table analysis is used for initial high-level characterization of each System Service for each image. The pivot table enables the method-level feature vector data to be aggregated into service-level statistics. Going back to the *NetworkManagementService* example used earlier, the aggregate feature data for various AOSP versions is displayed in Figure 4.3.

From this top-level graphical representation, it’s immediately evident that the number of AIDL

Table 4.1: Case study images.

Image	Android version	AOSP Baseline (if applicable)
AOSP jdq39	4.2.2	N/A
AOSP ktu84p	4.4.4	N/A
AOSP lmy48m	5.0	N/A
AOSP lrx21o	5.0.1	N/A
AOSP lrx22g	5.0.2	N/A
AOSP lrx22c	5.1.1	N/A
AOSP mra58n	6.0	N/A
BLU Neo4.5	4.2.2	jdq39 4.2.2
CyanogenMod 11-20150901	4.4.4	ktu84p 4.4.4
CyanogenMod 12.1-20151121	5.1.1	lrx22c 5.1.1
FireOS 32.4.6.5*	4.4.4	—
FireOS 37.5.2.2	5.0.2	lrx22g 5.0.2
LG D855PC10C_00	4.4.2	ktu84p 4.4.4
LG VS980	5.0.2	lrx22g 5.0.2
LG LS991ZV6_00	5.1	lrx22c 5.1.1
MotoX LXE22.46-11	5.0	lmy48m 5.0
SamsungEdge G925FXXU2COH8	5.1.1	lrx22c 5.1.1
SamsungNote8 N5110UEU2CNE2	4.4.2	ktu84p 4.4.4
SamsungS4 I9505XXUHOB7	5.0.1	lrx21o 5.0.1
SamsungS5 ATT G900AUCU1ANCE*	4.4.2	—
SamsungS5 Sprint G900PVPU1ANCB*	4.4.2	—
Xiaomi MIUI V7.0.5.0.KXDMICI	4.4.4	ktu84p 4.4.4

*Static analysis errors precluded full processing and analysis.

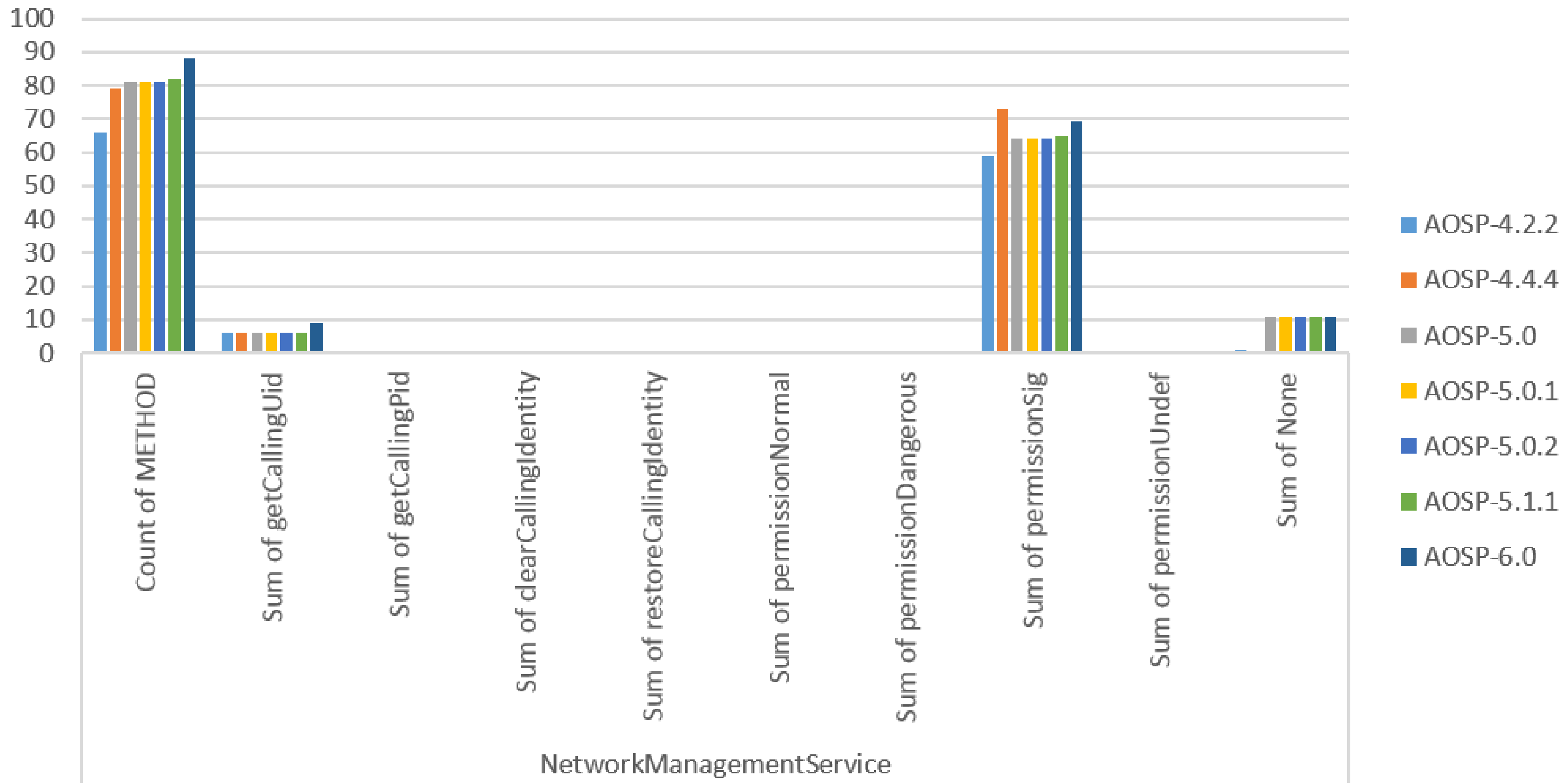


Figure 4.3: Count vs. access control feature for AIDL methods in various AOSP versions of NetworkManagementService.

methods in *NetworkManagementService* has steadily grown from Android version 4.2.x through 6.0. Also, some methods check the caller's identity by way of *uid*, but the vast majority check if the caller has been granted a *signatureOrSystem*-level permission, even those that were added by Google in newer releases. A few in the later releases of Android also have no access controls, a secondary feature, $noAC$, derived by checking the state of the others. Notably, the summary clearly shows what was manually confirmed earlier: *none* of the methods contain blocks with elevated privileges (i.e., no *clearCallingIdentity-restoreCallingIdentity* blocks), and *none* rely on *normal*- or *dangerous*-level permissions. Thus, the pivot analysis of the feature vector data has quickly and clearly revealed key access control characteristics of the service, across a couple of years of successive Android releases.

The PQFL and pivot functionality allows this analysis to be accomplished for any one or more services in any one or more images, just by placing the appropriate CSVs in the import directory structure. Appendix E shows the complete results for all services analyzed in AOSP images corresponding to Android versions 4.4.2, 4.4.4, 5.0, 5.0.1, 5.0.2, 5.1.1, and 6.0.

This baseline characterization provides insights into the nature of the security design of each System Service. In the case of *NetworkManagementService*, it's clear that the designers intended the service be used only by apps trusted to have *signatureOrSystem*-level privileges. In contrast, a service such as *LocationManagerService* is protected by predominately *dangerous*-level permissions, since location is considered sensitive privacy data, but necessary for 3rd-party apps if the user approves. In addition, as shown by the data in Appendix E, *LocationManagerService* performs 13-16 (depending on Android version) privileged operations, based on the count of *clearCallingIdentity* and *restoreCallingIdentity* features.

Many other insights can be gained from the feature vector data and the endless ways to analyze it. By including the raw feature vector data in a database, complex queries suited to the needs of the security investigator may be applied. For AOSP images, exploration of the feature set data can help guide the investigator through the source code using Android source code browsers such as

GrepCode[4] or AndroidXRef[5]. The feature vector database contributed by this work and its powerful analysis capabilities can thus be used to answer the question “what is the type and nature of access controls in System Services?”

4.4.3 Differential Analysis

A pivot analysis is also useful for a differential analysis using the feature vector difference CSVs. Figure 4.4 shows such an analysis for the SamsungS4 I9505XXUHOB7 5.0.1 image as it compares with the AOSP lrx21o 5.0.1 baseline. Here we can quickly see which new and existing services were added or modified, as well as the relative scope of the customization.

The database allows more detailed analysis to be performed as well. As one example of many possibilities, Table 4.2 depicts the results of an investigation of the same Samsung image to find all new or modified AIDL methods that enforce access control with a *dangerous*-level permission. This analysis quickly reveals 13 methods spread over 3 services that meet this criteria.

[4] http://www.grepcode.com/
[5] http://www.androidxref.com/

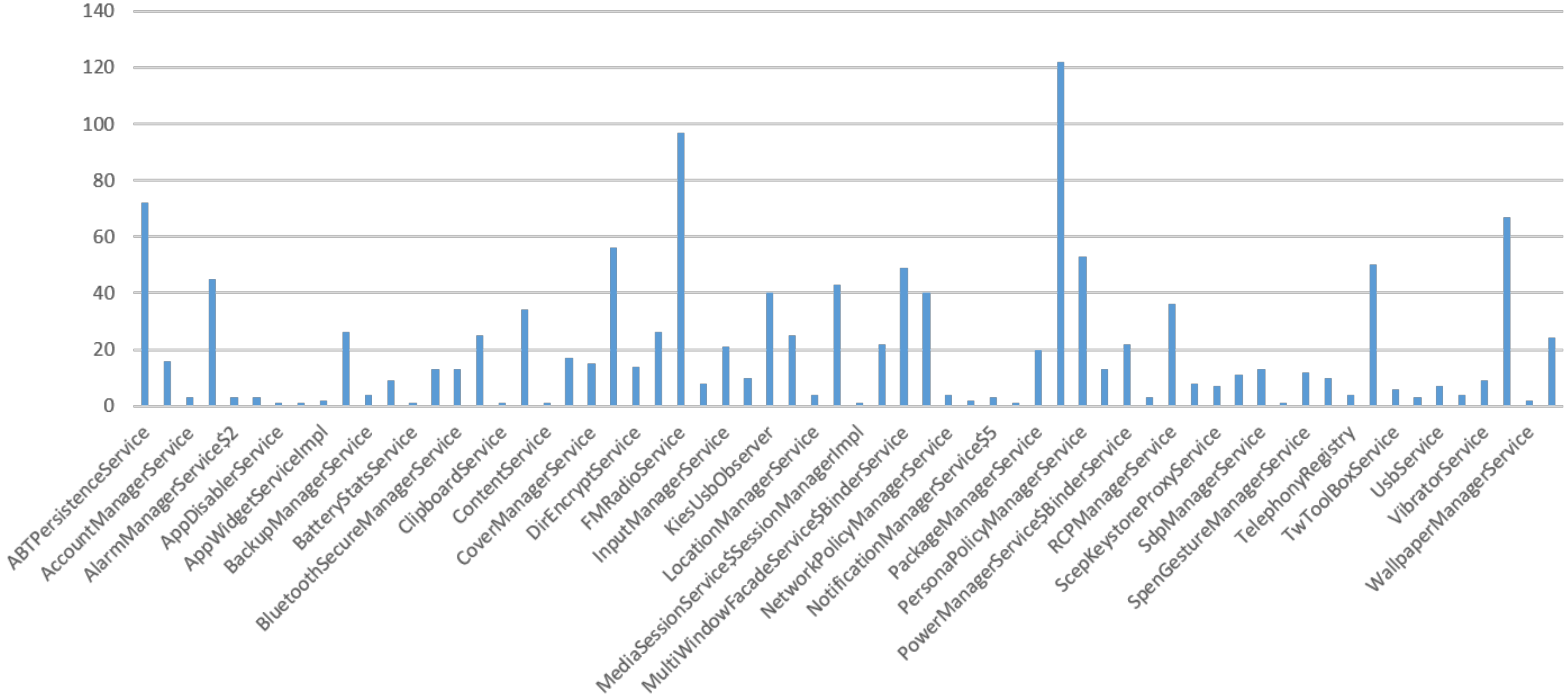


Figure 4.4: Pivot chart showing count vs. service name for AIDL methods added or changed in SamsungS4 I9505XXUHOB7 5.0.1 image as compared with AOSP lrx21o 5.0.1 image.

Table 4.2: SamsungS4-5.0.2 System Services containing new methods protected by dangerous-level permissions.

Criteria:
permissionNormal = *All*
permissionDangerous = 1
permissionSig = *All*
permissionUndef = *All*
None = *All*

	Service		
Method	**Activity Manager Service**	**Bluetooth Manager Service**	**Network Management Service**
disableKeyguard(Landroid/os/IBinder;Ljava/lang/String;)V	1		
dismissKeyguard()V	1		
enableGatt()Z		1	
getAccessPointNumConnectedSta()I			1
getAccessPointStaList()Ljava/lang/String			1
getLEAddress()Ljava/lang/String		1	
putLogs(Ljava/lang/String;)V		1	
readWhiteList()I			1
reenableKeyguard(Landroid/os/IBinder;)V	1		
setAccessPointDisassocSta(Ljava/lang/String;)I			1
setMaxClient(I)I			1
setTxPower(I)I			1
wps_ap_method(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String			1

As with the characterization analysis, there are endless possibilities for querying, analyzing and manipulating the feature data, depending on the needs of the researcher. As an indication of the extent of vendor modifications, Appendix F contains the results of a pivot analysis of all 8,037 added and modified System Service methods from 12 vendor images that were compared to their closest AOSP counterpart. These images contain an average of 670 added or modified AIDL methods, with a high value of nearly 1600 additions in the Samsung images.

These example comparison analyses represent a few of the many ways that the second two research questions are answered using the methodology described here. Using the database and analysis tools, the security researcher can very quickly determine, “has a vendor customized System Services for their devices and, if so, how?” and “if present, how do the customizations compare with a known baseline, AOSP?”

4.4.4 Method-level Evaluation

Feature vector analysis can only go so far in yielding security insights about access control and vendor customizations. Its purpose is to enable the researcher to gain a high-level understanding quickly and then be able to easily focus in on specific areas of interest. Without the aid of the various feature vector analyses described above, and the ability to quickly explore the entire System Services dataset, investigators must resort to time-consuming static analysis or combing through source code or decompiled JARs.

Once focused in on specific areas of interest, full and complete evaluation depends on traditional methods such as source code or bytecode review and actual testing. This section provides examples of how the feature vector analysis enabled rapid discovery of some problematic conditions in vendor customizations.

The focus now turns to specific methods of three selected images in the dataset: *LG VS980 5.0.2* (hereafter referred to as *LG-5.0.2*), *SamsungS4 I9505XXUHOB7 5.0.1* (*S4-5.0.1*), and *MotoX LXE22.46-11*

5.0 (*MotoX-5.0*). These were chosen for method-level evaluation because actual devices running these images were available for testing and verification of hypotheses made from the feature vectors.

Table 4.3 shows the result of querying the database for a count of all methods in each service that have the potential to be accessible by 3^{rd}-party apps. Based on experience, methods that invoke **`getCallingUid()`** do so in order to check whether the caller has Linux system privileges (i.e, *uid* of 0, 1000, or 2000). Also, as discussed earlier, *signatureOrSystem*-level permissions are not obtainable by 3^{rd}=-party apps. Thus, in terms of the feature set, the database can be queried for all records that match the following criteria:

$$f_{thirdPartyMethod} = f_{getCallingUid=0} \cap f_{permissionSig=0}$$

.

Table 4.3: Count of new 3^{rd}-party-accessible methods vs. service for three test images.

Service	Image		
	LG-5.0.2	**MotoX-5.0**	**S4-5.0.1**
ABTPersistenceService			70
AccessibilityManagerService			16
ActivityManagerService	39		23
ActivityManagerService$GmemBinder	30		
AlarmManagerService$2			1
AlarmManagerServiceExt$SyncScheduler$TrafficAnalyzer$2			3
AppDisablerService			1
AppOpsService	1		1
AudioService			21
BackupManagerService	1		1
BarBeamService			4
BluetoothManagerService	5		7
BluetoothSecureManagerService			10
ClipboardExService			22
ClipboardService			1

continued…

Table 4.3: Count of new 3rd-party-accessible methods vs. service for three test images.

Service	**Image**		
	LG-5.0.2	**MotoX-5.0**	**S4-5.0.1**
CommonTimeManagementService	1		
ConnectivityService	25		15
ContentService			1
ContextAwareService			17
CoverManagerService			15
DataSchedulerService	3		
DeviceManager3LMService	94		
DevicePolicyManagerService			27
DirEncryptService			7
DisplayManagerService$BinderService			21
FastDownloadService	18		
FMRadioService			96
HarmonyEASService			7
InputManagerService	1		15
InputMethodManagerService			9
KiesUsbObserver			39
KtUcaService	25		25
LGEncryptionService	1		
LocationManagerService			4
LockSettingsService			14
MediaSessionService$SessionManagerImpl			1
MHPService	126		
MountService			15
MultiWindowFacadeService$BinderService			49
NetworkManagementService	15	6	10
NetworkPolicyManagerService			1
NotificationManagerService$5			2
NotificationManagerService$8			1
OemExtendedApi3LMService	9		
PackageManagerService	5	1	9
PersonaManagerService			19
PluginManagerService$PluginBinder			13
PowerManagerService$BinderService	2	1	10
PowerSaving3LMService	5		
QuickConnectService			2

continued…

Table 4.3: Count of new 3rd-party-accessible methods vs. service for three test images.

Service	**Image**		
	LG-5.0.2	**MotoX-5.0**	**S4-5.0.1**
RCPManagerService			4
ReactiveService			6
ScepKeystoreProxyService			3
SContextService			11
SdpManagerService			1
SearchManagerService			1
SmartCoverService	4		
SpenGestureManagerService			12
StatusBarManagerService		2	7
TelephonyRegistry	1		
ThemeIconManagerService	4		
TimaService			44
TwToolBoxService			6
UsageStatsService$BinderService			1
UsbService			2
UserManagerService			2
VibratorService			9
VoIPInterfaceManager			3
VzwConnectivityService		3	
VzwLocationManagerService	11		
WallpaperManagerService			1
WiFiAggregationService	6		
WiFiOffloadingService	28		
WindowManagerService	12	2	20
Total	**472**	**15**	**758**

This focused feature dataset can be used to guide additional detailed analysis. For example, focusing in on *NetworkManagementService*, it can be seen that *LG-5.0.2* contains 15 new accessible methods, while *MotoX-5.0* and *S4-5.0.1* contain 6 and 10 new methods, respectively. A further query of the feature vector database yields the following list of actual method names, argument types and return types. These are determined during static analysis and are shown as Java object types (e.g., `Ljava/lang/String`) and symbols (i.e., `V` for void, `Z` for boolean and `I` for integer).

- *LG-5.0.2*:
  - `SKTCatsPortForwarding(Ljava/lang/String;)V`
  - `acceptPacket(Ljava/lang/String;)V`
  - `addUpstreamV6Interface(Ljava/lang/String;)V`
  - `dropPacket(Ljava/lang/String;)V`
  - `getRouteList_debug(Ljava/lang/String;)V`
  - `packetList_Indrop()Z`
  - `packetList_Indrop_view()V`
  - `registerObserverEx(Landroid/net/INetworkManagementEventObserverEx;)V`
  - `removeUpstreamV6Interface(Ljava/lang/String;)V`
  - `resetPacketDrop()V`
  - `runShellCommand(Ljava/lang/String;)V`
  - `setDhcpv6Enabled(ZLjava/lang/String;)V`
  - `setInterfaceAlias(Ljava/lang/String;Ljava/lang/String;)V`
  - `setTcpWindowScaling(Z)V`
  - `unregisterObserverEx(Landroid/net/INetworkManagementEventObserverEx;)V`
- *MotoX-5.0*:
  - `addUpstreamV6Interface(Ljava/lang/String;)V`
  - `getSapAutoChannelSelection()I`
  - `getSapOperatingChannel()I`
  - `removeUpstreamV6Interface(Ljava/lang/String;)V`
  - `runIpLogCmd(Ljava/lang/String;)I`
  - `setChannelRange(III)V`
- *S4-5.0.1*:
  - `addUpstreamV6Interface(Ljava/lang/String;)V`
  - `appendInterfaceToLocalNetwork(Ljava/lang/String;Ljava/util/List;)V`
  - `getAccessPointNumConnectedSta()I`
  - `getAccessPointStaList()Ljava/lang/String`
  - `readWhiteList()I`
  - `removeUpstreamV6Interface(Ljava/lang/String;)V`
  - `setAccessPointDisassocSta(Ljava/lang/String;)I`
  - `setMaxClient(I)I`
  - `setTxPower(I)I`
  - `wps_ap_method(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String`

It is important to note that the methods included here constitute a *superset* of those that may be accessible to 3rd-party apps. This is because the current feature set does not attempt to capture everything about the call chain between an entry point and the resources ultimately accessed.

Some entry points may depend on access controls further down the call chain or access controls not captured by the feature set. For example, Samsung's custom *ABTPersistanceService* includes an AIDL method *getDeviceId()* which appears to have no access control from a feature vector point of view. However, manual testing revealed that this method in fact calls a private method `g()`, which requires the caller to have a Linux *uid* of 1000. As it turns out, all of the AIDL methods in this service are protected in the same way, using `g()`. Addressing this in the *FeatureExtraction* software is possible, but would require additional static analysis time to follow the call path to find access control at deeper levels. This ends up being a trade-off between static analysis complexity and extra testing time to eliminate false positives.

For actual method-level testing, a test app, *ServiceApiTest*, was developed and used in conjunction with each physical device to verify that the API methods are indeed exposed to 3rd-party apps. Using reflection, each candidate method is called with the appropriate arguments and return types, which are known from the static analysis and indicated in the lists above. For example, the *MotoX-5.0* method *setChannelRange()* has three integer arguments (denoted by `III` above), and is return type void (denoted by `V`).

Listing 4.2 shows the key portion of the code that can be used to invoke any public method in any service running on the device. To call a method under test, *SystemService.getServiceProxyObject()* is first used to get a handle to the service that contains the method. Then a reference to the method is found in the proxy class of the interface using *getMethod()* and the method is invoked. Results from the invocation are written to *logcat* and captured.

As an example, representative *logcat* output (for *setChannelRange()*) is shown in Listing 4.3. This invocation is successful as evidenced by the status messages from the service. An example of an unsuccessful invocation is shown in Listing 4.4, which indicates that Samsung's *getDeviceId()* is protected by access controls deeper in the call chain than visible with the current feature set and static analysis approach.

Listing 4.2: App code to call test methods by reflection.

```
static final String LOG_TAG = "ServiceApiTest::NMS";

static String serviceInterfaceName = "android.os.INetworkManagementService";
static String serviceProxyName = "android.os.INetworkManagementService$Stub$Proxy";
static String serviceManagerRegistrationName = "network_management";

public static void testMethodCall(Context context) throws ClassNotFoundException,
        SecurityException, NoSuchMethodException, IllegalArgumentException,
        IllegalAccessException, InvocationTargetException,
        InstantiationException, IOException {

    Class proxyClass = Class.forName(serviceProxyName);
    Object serviceProxyObject = SystemService.getServiceProxyObject(serviceInterfaceName,
            serviceProxyName, serviceManagerRegistrationName);
    Log.d(LOG_TAG, "Returned object = " + serviceProxyObject.toString());

    try {
        // Following depends on the method to be invoked
        //Object result[] = null; // use this if method returns array
        Object result = null;       // use this otherwise (change below too)

        // Following depends on the method to be invoked
        Method testMethod =
                proxyClass.getMethod("setChannelRange", int.class, int.class, int.class);
                // ‘method name ‘arg type(s)
        Log.d(LOG_TAG, "Found method = " + testMethod.toString());

        //result = (Object[])testMethod.invoke(serviceProxyObject); // use this if method returns array
        result = testMethod.invoke(serviceProxyObject, 2, 3, 5);      // otherwise use this
            (change above too)
        // ‘arg(s)
        Log.d(LOG_TAG, "Result = " + result); // ObjectUtil.serializeObjectToString(result));
    }

    catch (NoSuchMethodException e) {
        Log.e(LOG_TAG, e.toString());
    }
```

Listing 4.3: logcat output from successful invocation of setChannelRange() method in customized MotoX-5.0 NetworkManagementService.

```
D/ServiceApiTest::MA(25369): Invoking test method...
D/ServiceApiTest::SS(25369): Returned object = android.os.
    INetworkManagementService$Stub$Proxy@268a3266
D/ServiceApiTest::NMS(25369): Returned object = android.os.
    INetworkManagementService$Stub$Proxy@268a3266
D/ServiceApiTest::NMS(25369): Found method = public void android.os.
    INetworkManagementService$Stub$Proxy.setChannelRange(int,int,int) throws android.os.
    RemoteException
D/NetworkManagementService(  933): Set SAP Channel Range
D/        (  343): CMD INPUT  [ set setchannelrange= 2   3   5][256]
E/        (  343): Cmd: setchannelrange Argument : 2   3   5
D/        (  343): cmd=setchannelrange, Val: 2   3   5, INI:0
D/        (  343): Updated:setchannelrange= 2   3   5
D/        (  343):
D/        (  343): CMD OUTPUT [success]
D/        (  343): len :7
D/        (  343):
D/ServiceApiTest::NMS(25369): Result = null
D/ServiceApiTest::MA(25369): ...finished.
```

Listing 4.4: logcat output from unsuccessful invocation of getDeviceId() method in customized S4-5.0.2 ABTPersistanceService.

```
D/ServiceApiTest::MA( 2286): Invoking test method...
D/ServiceApiTest::SS( 2286): Returned object = com.absolute.android.persistence.
    IABTPersistence$Stub$Proxy@274b8d19
D/ServiceApiTest::ABTPS( 2286): Returned object = com.absolute.android.persistence.
    IABTPersistence$Stub$Proxy@274b8d19
D/ServiceApiTest::ABTPS( 2286): Found method = public java.lang.String com.absolute.android.
    persistence.IABTPersistence$Stub$Proxy.getDeviceId() throws android.os.RemoteException
E/ServiceApiTest::MA( 2286): java.lang.reflect.InvocationTargetException
W/System.err( 2286): java.lang.reflect.InvocationTargetException
W/System.err( 2286):         at java.lang.reflect.Method.invoke(Native Method)
W/System.err( 2286):         at java.lang.reflect.Method.invoke(Method.java:372)
W/System.err( 2286):         at com.ratazzi.serviceapitest.ABTPersistenceService.
    testMethodCall(ABTPersistenceService.java:43)
W/System.err( 2286):         at com.ratazzi.serviceapitest.MainActivity.onCreate(
    MainActivity.java:25)
W/System.err( 2286):         at android.app.Activity.performCreate(Activity.java:6289)
W/System.err( 2286):         at android.app.Instrumentation.callActivityOnCreate(
    Instrumentation.java:1119)
W/System.err( 2286):         at android.app.ActivityThread.performLaunchActivity(
    ActivityThread.java:2646)
W/System.err( 2286):         at android.app.ActivityThread.handleLaunchActivity(
    ActivityThread.java:2758)
W/System.err( 2286):         at android.app.ActivityThread.access$900(ActivityThread.java
    :177)
W/System.err( 2286):         at android.app.ActivityThread$H.handleMessage(ActivityThread.
    java:1448)
W/System.err( 2286):         at android.os.Handler.dispatchMessage(Handler.java:102)
W/System.err( 2286):         at android.os.Looper.loop(Looper.java:145)
W/System.err( 2286):         at android.app.ActivityThread.main(ActivityThread.java:5942)
W/System.err( 2286):         at java.lang.reflect.Method.invoke(Native Method)
W/System.err( 2286):         at java.lang.reflect.Method.invoke(Method.java:372)
W/System.err( 2286):         at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(
    ZygoteInit.java:1400)
W/System.err( 2286):         at com.android.internal.os.ZygoteInit.main(ZygoteInit.java
    :1195)
W/System.err( 2286): Caused by: java.lang.SecurityException: Not authorized to access ABT
    Persistence Service
W/System.err( 2286):         at android.os.Parcel.readException(Parcel.java:1540)
W/System.err( 2286):         at android.os.Parcel.readException(Parcel.java:1493)
W/System.err( 2286):         at com.absolute.android.persistence.IABTPersistence$Stub$Proxy.
    getDeviceId(IABTPersistence.java:630)
W/System.err( 2286):         ... 17 more
```

4.4.5 Results

Because it was done manually in this work, testing of actual methods was limited to a sample. In total, 81 methods, selected from across the three images were tested using the methodology described above. Services successfully tested include:

- *LG-5.0.2*: 54 methods selected from *NetworkManagementService*, *InputManagerService*, *BackupManagerService*, *AlarmManagerService*, *CCModeService*, *ClipboardService*, *ConnectivityService*, *LGEncryptionService*, *MHPService*, and *WiFiOffloadingService*.

- *S4-5.0.1*: 10 methods from *ABTPersistanceService*, *AccessibilityManagerService*, *WindowManagerService*, *AudioService*, and *ClipboardService*.
- *MotoX-5.0*: 17 methods from *NetworkManagementService*, *PowerManagerService*, *StatusBarManagerService*, *VzwConnectivityService*, and *WindowManagerService*.

Detailed test results and test notes are located in Appendices G, H, and I for *LG-5.0.2*, *S4-5.0.1*, and *MotoX-5.0*, respectively. Overall findings are summarized as follows:

- All 15 of the new methods in LG's customized *NetworkManagementService* identified as 3rd-party-accessible by the feature vector data were confirmed to be so. Of particular interest were methods that involved enumeration or manipulation of the device's network interfaces, routing tables, and firewall (*iptables)*. In addition, LG's use of non-*signatureOrSystem* permissions in this service contrasts with AOSP's *exclusive* use of them.
- LG's customized *ConnectivityService* includes several "getter" methods that have no permission requirement. This contrasts with AOSP's use of `ACCESS_NETWORK_STATE` to protect network state information. Also, several "setter" methods, such as *setDataBlock()* and *setRoamingDataEnabled_RILCMD()* had no permission requirement, which contrasts with AOSP's typical use of `CHANGE_NETWORK_STATE` to protect changes to network settings.
- LG includes a custom service, *MHPService* (related to mobile hotspot settings) which contains 126 3rd-party-accessible methods, all with no access control features. Of the five selected for testing, all were confirmed to have no access control and were invocable. All five changed important device settings or returned sensitive information. Examples are enabling and disabling the mobile hotspot, getting and setting the WiFi Protected Access (WPA) keys, and getting and setting the mobile hotspot Service Set Identifier (SSID). At a minimum, to be consistent with AOSP, these methods should be protected with

CHANGE_NETWORK_STATE or ACCESS_NETWORK_STATE. More likely, they are methods that should only be accessible to bundled LG system apps and thus should be protected with CONNECTIVITY_INTERNAL or a custom *signatureOrSystem*-level permission.

- LG includes a custom service, *WiFiOffloadingService* which includes 24 methods with no access controls. *disableWifi()* was confirmed to be invocable and change device network state. The lack of access control in this service is inconsistent with AOSP's typical use of CHANGE_NETWORK_STATE or ACCESS_NETWORK_STATE to protect network information and state changes.
- Samsung's customized *AccessibilityManagerService* contains 16 new methods with no access control features. Of the 3 tested, two has no access controls and appeared to make changes to the device's display. The third requires the system permission REBOOT. The permission is enforced elsewhere, explaining the lack of access control features.
- Samsung's *AudioService* contains 21 new methods, 2 of which have the *getCallingGid* feature, while the others have no access control features. Three of these were tested and appear to have effect, as the log reports activity in *VolumePanel*.
- Motorola's *NetworkManagementService* contains 6 new methods that are potentially available to 3rd-party apps. Five of these use *normal*- or *dangerous*-level permissions and 1 has no access control features. All 6 were tested. Execution of 2 seemed to be blocked by SELinux denials, while the others were successful. Access control for all 6 methods is inconsistent with AOSP's use of *signatureOrSystem*-level permissions in this service, and the unprotected method, *runIpLogCmd()* should be considered for access controls, since it passes raw commands to a native daemon and is thus high risk for misuse.
- Motorola's *PowerManagementService* contains one accessible method with no reported access control features. It was successfully invoked during testing.
- Motorola's *StatusBarManagerService* contains two methods without access control features. These were confirmed to change the state of the device's display by revealing or hiding a search widget.

4.5 Other Applications

Generically, the feature extraction, characterization, and comparison methodologies introduced here may have applicability to analyzing access controls in systems other than Android. Overall, the feature vector-based approach is most useful when analyzing presence or absence of access control at interfaces between components of a larger system. Systems that have these characteristics include modular and client-server architectures, especially those that have a well-defined common security framework that should be used when building extensions.

An example of such a system are Linux kernels with a common security framework such as Linux Security Modules (LSM) [58]. LSM provides kernel developers with a general-purpose framework for implementing standard access controls in future modules and extensions.

The standardization afforded by LSM is analogous to the insights gained in this work about Android's relatively standard use of specific access control patterns throughout the system and specifically in System Services. Just as these insights enabled the definition of a feature set applicable to many System Services, the LSM API can be translated to a feature set. Appropriate static analysis tools can then be applied to source or binary representations of the kernel or kernel modules to examine how, where, and if standard access controls are implemented. Moreover, kernel changes or customizations can be assessed via differential analysis of feature vector databases from different kernel versions, just as was done in this work for vendor customizations of Android System Services.

Client-server web applications represent another class of systems that may benefit from a feature set-based approach. Implementations of these systems have also benefitted from standard security frameworks such as [59] and [60]. APIs describing the framework can be translated into a feature set which can then be extracted from code using static analysis tools. Various efficient analyses and comparisons are then facilitated by the resulting feature vector database.

4.6 Limitations and Future Directions

Some images were not able to be fully analyzed due to errors in unpacking or during static analysis. It is believed that this is due to variances in vendor image formats, non-standard files, deodex tool limitations, and the usual limitations of static analysis. Unfortunately, this means that in some cases, not all System Services are represented in the feature vector files. To address this, more robust unpacking and deodexing tools must be developed. Nevertheless, even partial feature vector analysis can provide meaningful insight into portions of a vendor's propietary image.

More complex features would also improve the fidelity and usefulness of the results. For example, adding features that describe how the return value of *getCallingUid* and *getCallingPid* is used would help in understanding access controls that require the caller to have specific Linux-based credentials. Also, feature extraction from deeper along the call chain would reduce the number of methods thought to be without access controls, and in turn reduce the amount of verification testing.

Finally, common analysis macros could be developed to avoid having to manually build queries over the feature vector databases. These would be tailored to the domain of Android access control, as opposed to the generic data analysis tools available in Microsoft®Excel.®

4.7 Conclusion

This chapter described a feature vector-based approach to studying and understanding access controls in Android. The method was used to develop an interactive database of access controls in the System Services of 19 Android images. Seven of these are AOSP images and are used as a baseline in a case study to analyze the changes introduced by the vendors of the other 12. This differential analysis is captured in a second interactive database. Selected changes from three of the vendor images were further studied on actual devices, in order to confirm the feature vector data, its utility and show that the methodology can be used practically to find potential shortcomings in vendor customizations. The case study revealed a number of issues and inconsistencies with the vendor code.

Chapter 5

Protecting Sensitive and Vulnerable Resources

> *Traditionally, we've thought about security and usability as a trade-off: a more secure system is less functional and more annoying, and a more capable, flexible, and powerful system is less secure. This "either/or" thinking results in systems that are neither usable nor secure.*
>
> - Bruce Schneier

Chapters 3 and 4 describe security analysis methodologies that provide insights into the inner workings of Android access controls, in both open source and vendor implementations. As the results show, the advantages of an open, modular system, whereby each resource is responsible for implementing its own access controls, can be offset by security problems arising from mistakes and inconsistencies. In particular, protections implemented by resources can be partially or completely nullified when changes to the platform architecture or vendor customizations invalidate the assumptions implicit in the original design. To address these situations, each and every resource must be reevaluated and redesigned every time the platform changes. If system designers and vendors use the methodologies of the previous chapters, then these mistakes can be fixed before the platform is deployed. However, once deployed, end-users have virtually no way of addressing any remaining problems unless the platform itself contains a flexible, generic means of isolating any vulnerable resource(s) from untrusted apps.

Many of the system insights gained during the course of work described in earlier chapters relate to the mechanisms by which Android's modular system resources are requested, protected, and accessed. The knowledge gained from the systematic, top-down subject-object access control path evaluation, combined with that from the detailed method-level feature vector-based characterization and comparison led to new insights. These led to the realization that almost any current or future system resource could be transparently isolated from apps that may take advantage of access control shortcomings or, for critical apps, fall victim to poorly designed or malicious resources. This chapter describes the concept and design methodology that emerged from these realizations, and demonstrates through an actual implementation and case study how the novel approach is simultaneously efficient, effective, and consistent with Android's unique architecture and design tenets.

5.1 Introduction

Security and privacy compromises by malware and faulty apps is a persistent concern of smartphone users. While many users may not fully understand the technical aspects of security architectures, permissions, access control mechanisms, or measuring trust, most have no trouble articulating which high level objects, resources or capabilities they are most concerned with. For example, it's common to find users worried about how apps might misuse or leak their location, sensitive data such as personal contacts, or personally-identifiable information (PII) like phone number and International Mobile Station Equipment Identity (IMEI). In response to this, numerous solutions to address these concerns have been proposed, and many of these involve some form of virtualization combined with access control to isolate untrusted applications.

Although every approach to isolation has its own unique strengths and weaknesses, all include trade-offs in terms of sharing and communication. In Android's open architecture, where resource sharing and inter-process communication (IPC) are fundamental to the platform's basic operation and usability, careful attention must be paid to fully understanding how a particular isolation boundary impacts the system's functionality and performance. If this trade-off is not considered at the outset of a design, significant performance, usability, and functionality issues can arise. Countering these negative side-effects requires designers to overcome challenging system problems, typically resulting in substantial modifications to the operating system, and significant second-order complexities not directly related to the initial security goals. These problems are especially prevalent in general-purpose designs that attempt to provide isolation containers for entire apps or virtual phones, without the benefit of *a priori* knowledge of specific threat(s) or end-user security goals.

This chapter introduces *PINPOINT*, a resource isolation strategy that forgoes general-purpose solutions in favor of a lightweight approach that addresses only specific end-user security goals. By addressing the end-users' stated security goals and no more, PINPOINT yields an effective result using only the minimum amount of isolation. This significantly reduces or even eliminates the negative side-effects that inevitably emerge when large parts of Android's open, shared architecture are isolated. Because isolation of resources can occur at many places within the system architecture, and with varying degrees of granularity, the chapter begins with a discussion of the tradeoffs in the design space in Section 5.2. This is followed by Section 5.3, which contains a high-level description of the chosen concept. Section 5.4 describes a case study whereby we implemented the PINPOINT concept as a lightweight *hypo*visor[1] within Android's Context Manager, facilitating isolation of any System Service. Since resources available to apps are typically presented as System Services, this case study implementation addresses a wide range of practical security and privacy problems.

5.2 Design Space

In a layered architecture such as Android, isolation of a particular resource could be accomplished at many levels, from the most abstract all the way down to the hardware device itself. Each of

[1] See Section 5.3.1 for an explanation of this new term.

these alternatives has both advantages and disadvantages in terms of its impact on the system and the end-user's experience. With this in mind, this section contains an analysis of the alternatives within the design space such that we can find the best solution given the metrics deemed important. In this work, that includes solutions that are effective for specific threats or classes of threats (as required by the end-user), while remaining transparent to developers, negligible in their performance impacts, free of complex changes to the system, and low-cost to the end-user in terms of usability and convenience. As we traverse the design space, we consider the following qualitative aspects as our measures of merit for each design alternative:

1. *Range of threats*: does the design address a wide range and variety of threats?
2. *Adequacy*: is the design adequate in containing the threats it is intended to address?
3. *Isolation "size"*: is the isolation comprehensive, resulting in a small attack surface (few things shared)?

Together, these measures relate to the notion of *effectiveness*, while an aggregation of the following contribute to *efficiency*:

1. *Complexity*: does the design require far-reaching or complex modifications to the Android system in order to restore basic platform functionality or compensate for loss of end-user usability/convenience?
2. *Transparency*: does the design require special considerations on the part of the application developer, i.e., do apps need to be modified to function?
3. *Performance*: does the design have a significant impact on overall device performance in terms of start-up, application launch, user interface response, etc.?
4. *Usability and convenience*: will the end-user face significant concessions in terms of usability and/or convenience in order to realize their security goals?

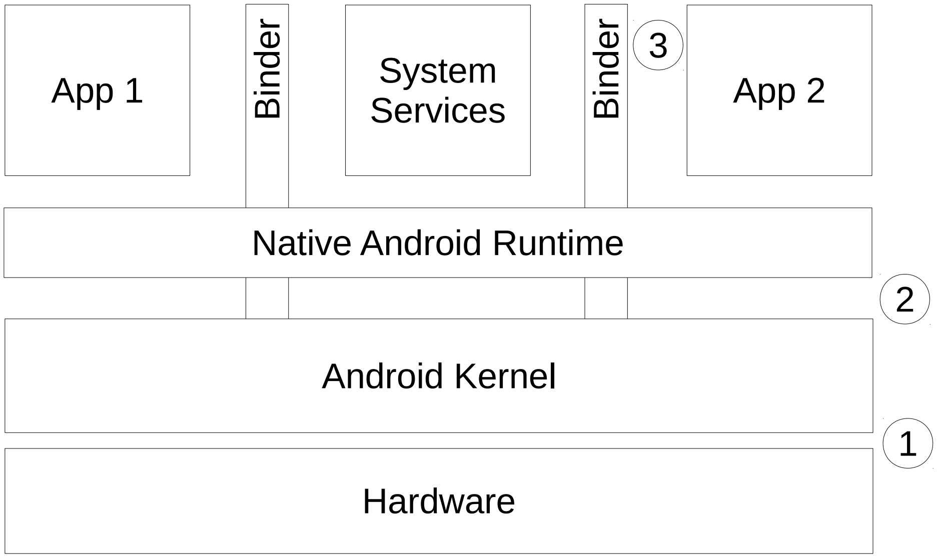


Figure 5.1: Simplified Android architecture.

5.2.1 Tradeoff analysis

Consider the simplified Android architecture diagram of Figure 5.1. Here we see that all applications and high-level system services share a single runtime, with inter-app communication facilitated by *Binder*, Android's lightweight IPC subsystem [61]. Now consider the situation where we don't trust App 2 and wish to isolate it in some way. This diagram now represents our tradespace, where we can consider the overall ramifications of different approaches to isolating App 2, in terms of the seven qualitative measures of merit introduced above.

First, we could totally isolate App 2 by running it on an entirely different instance of the kernel and operating system (i.e., "dual boot"), where the only thing shared between the two apps is the device hardware (Figure 5.1, location ①). Using a mechanism like the open source *BootManager* project [62], this approach provides a strong isolation between untrusted App 2, App 1, and the system services on the OS instance we wish to protect. However, this very strong isolation comes with a very high price. Usability and convenience suffer greatly since switching between apps involves a full reboot, and interactive sharing of data and resources becomes impossible. Also, as is common in Android, App 2 may depend on components of App 1 for full functionality, requiring separate copies of App 1 to be installed in both boot partitions. On the other hand, the isolation is very strong, and even if App 2 is able to exploit kernel vulnerabilities, it cannot escape its isolation and affect App 1.

Inserting a virtualization layer at location ① to enable simultaneous virtual machines (VM) is another approach that recent work has shown possible [63]. This is the “bare metal” or Type 1 hypervisor design that would increase convenience somewhat by precluding the need for full reboots to switch between apps. Isolation with a native hypervisor is still very comprehensive, presenting App 2 with a very small attack surface. However, many of the sharing and open communication tenets of the Android design are still broken and would require a significant amount of added complexity to restore.

If we instead allow the isolated app to share both the hardware, kernel and some native Android operating system resources with the trusted app (Figure 5.1, location ②), one major aspect of usability improves somewhat. Similar to Type 2 hypervisor architectures, the user no longer has to reboot the device to switch apps. While the shared kernel and native processes represent additional attack surfaces (App 2 can now use a kernel vulnerability to escape its isolation and attack App 1), it enables low-level sharing of raw resources and communication channels, and the design is still general enough to address a wide variety of common threats. Unfortunately, Android fundamentally assumes a common runtime, and breaking this assumption introduces many difficulties. To address these difficulties, one must undertake challenging and complex side projects, such as those described in [64] and [65], which both use different variations of this approach to isolation.

Another possibility would be to allow the apps to share the same hardware, kernel and runtime, but not share the entirety of a frequently-misused subsystem such as Binder (Figure 5.1, location ③). From results described in [66], where this approach was taken, we see that although the effectiveness is limited to threats promulgated through IPC, nasty system problems (and additional corrective complexity) that come with isolating the Android runtime have been avoided. Usability and convenience also benefit, since an isolated IPC subsystem likely has much less negative effect on performance than does concurrent instances of the Android runtime environment. App 2's attack surface is also greater, since it now has the opportunity to exploit vulnerabilities in aspects of the framework other than IPC. Again, whether this was a prudent trade-off depends on the end-user's expectations and the threats they need addressed.

As can be seen by the design alternative summary in Table 5.1, it's clear by now that things like complexity, usability, and the range of threats addressed appear to be strongly correlated to how much of the system is shared by the isolated app and how much is part of the isolation itself (isolation size). If large portions of the system are part of the isolation (i.e., not much is shared between trusted and untrusted apps), many threats are addressed, but usability drops off, countered only by a commensurate increase in complexity to fix things. Conversely, as more system components are shared among all apps, fewer threats are addressed, but usability suffers less and fewer complex system modifications are required to compensate. At this level of analysis, all designs are assumed to be adequate in addressing the threats they were designed to address, and transparent to apps (i.e., apps can run unmodified without crashing). Thus, we can summarize the more interesting aspects of the design tradespace by noting a few apparent correlations among several of our measures of merit:

1. Range of threats addressed increases as isolation size increases (i.e., the attack surface decreases).
2. Complexity increases (i.e., having to fix things broken by the isolation) as isolation size increases.
3. Usability and convenience decreases as isolation size increases.

Thus, when the security goal is to address a specific threat or threats, the best solution would be one that isolates only what's necessary and nothing more. In this way, all relevant threats are addressed without sacrificing any more usability than necessary and without increasing ancillary complexity (that which is not directly related to the security goals) any more than necessary. In

Table 5.1: Summary of Isolation Design Alternatives

Design	**Range of threats**	**Adequate**	**Isolation size**	**Complexity**	**Transparent**	**Performance**	**Usability/ convenience**
Dual-boot	Most	Yes	Large	High	Yes	Excellent	Low
Type 1 hypervisor	Most	Yes	Large	High	Yes	Excellent	Low+
Type 2 hypervisor	Many	Yes	Medium	Medium	Yes	Good	Medium
Isolated IPC	Few	Yes	Small	Low	Yes	Excellent	Good

[66], this mindset is what allows their approach to isolating IPC to be termed “lightweight.” In this work however, we consider this as a more general design principle that is potentially applicable to many different aspects of the Android Framework, not just IPC. The next section describes a design concept that embodies this mindset.

5.3 Design Concept

Having explored the tradeoffs related to the various approaches to achieving resource isolation in Android, it is now possible to establish a more concrete design concept that addresses security needs without sacrificing key features and convenience. As mentioned earlier, several previous efforts have used isolation to address the problem of untrusted apps having access to sensitive or private information. Some of these address specific types of data, such as location, while others look for more general solutions (see Chapter 6 for more details). Two approaches that influenced this work tremendously are Cells[65] and AirBag [64]. Cells leverages Linux Namespaces to allow multiple Android user spaces, or virtual phones, to run simultaneously on a single hardware platform. AirBag also leverages Linux Namespaces, but achieves isolation at the native runtime boundary.

As an isolation mechanism, Linux Namespaces have several traits which correspond well to the favorable characteristics of minimal isolation described in Section 5.2. Because of this, a systematic analysis of Linux Namespaces, detailed in Appendix J, was undertaken to better define these traits and their specific value to Android security. This analysis identified six key traits that have value to the goal of providing effective yet efficient security, and are summarized in Table 5.2. However, when comparing these benefits to the existing work that leverages Linux Namespaces, it becomes apparent that much of the positive value of Namespace isolation was not realized in these solutions. Understanding why is key to proposing a design concept that avoids this pitfall.

Looking at the Linux Namespace-based solutions in more detail, one finds that while effective for

Table 5.2: Summary of Namespace Traits and the Value to Android Security (see Appendix J for details).

Namespace Trait	Value to Android Security
Fine-grained isolation of specific resources	Tailored isolation environment for each application, addressing specific threat and/or user goal
Resource-centric isolation	Match user perspective on security; increase usability; simplicity
High efficiency	Negligible performance impact; design simplicity
Share-by-default	Preserve open system design; avoid breaking things unrelated to the isolated resource
Transparent to host and apps	System retains control over apps; apps run unmodified
Small footprint (files, memory)	Little impact on performance & resources; OTA updates

a broad range of threats, each introduces a number of significant challenges that must be addressed before the overall system can function anywhere near what was intended and/or acceptable to the user. For example, approaches that isolate untrusted apps in a separate runtime [64] or virtual phone [65] require special customization of many shared hardware drivers, such as those of the framebuffer and graphics processing unit (GPU), resulting in a great deal of additional complexity which may have little or nothing to do with the end-user's primary security objectives. In fact, in order to work at all, these designs require duplication of a number of system processes and resources that may also have little or nothing to do with security objectives. In these architectures, fundamental Android features such as IPC, which have a direct relationship to usability and convenience, become problematic due to isolation of the Android runtime. Compensating for these negative impacts can result in significant additional complexity in the design. In the case of IPC, this requires the addition of special communication channels to partially bridge the isolated runtime with the rest of the system.

In spite of the practical limitations of these designs, their use of Linux Namespaces is intriguing, as Namespaces are generally considered to be a lightweight and efficient form of isolation. In traditional Linux systems, Namespaces are a useful tool for abstracting system resources so that different applications have an isolated view of the resource. Namespaces have been used to facilitate checkpoint/restart in high-performance computing (HPC) [67,68] and as the basis for Linux Containers (LXC) to provide efficient security and resource containers [69]. The typical uses of these technologies require only *certain* resources to be virtualized, and thus do not require the creation of multiple instances of the operating system kernel. This allows higher density than would be possible with traditional virtual machines. This advantage of container-based virtualization has been long-recognized and put to good use in HPC and cloud environments [70,71].

Why then haven't Linux Namespace-based approaches to application isolation in Android been all that practical, efficient, or successful at being accepted into the mainstream? The answer lies in the fact that Android applications do not access system resources directly, but rather through the layers of abstraction presented by the Android Framework, which runs on top of the Linux system. Thus, the use of Linux Namespaces in Android presents an isolated view of resources *to the entire Android Framework*, rather than to just individual applications. Because the Framework was not designed exist in multiple instances of the global namespace, many problematic side-effects arise. These side-effects and the additional complexity needed to overcome them, negate the positive traits that Namespaces have for their traditional applications.

Realizing the benefits of namespace-based isolation, without causing the problems and side-effects seen by their direct use is the goal of this design. Thus, the concept introduced here is to adopt the Linux Namespace *concept* of fine-grained isolation, but do so with a new implementation that is better suited to the resource abstractions of the Android Framework. In essence, this work seeks to move the point of isolation as close as possible to the object(s) requiring isolation, based on the stated security goals, while allowing continued sharing of everything else. Put another way, the goal is to realize the benefits of lightweight isolation in Android by identifying strategic locations where Linux Namespace *concepts* can be implemented. As we will show, the result is a simpler implementation free of problems and far-flung platform side-effects. When specific security goals are taken into account, the result can be just as effective as general-purpose solutions.

5.3.1 Hypervisor vs. Hypovisor

Before describing the high-level concept for PINPOINT, a brief aside on terminology is warranted.

In IBM's System/360 Operating System, the *supervisor* was a program that had complete control over everything running on the system. Later, and in most other systems, this program was known as the *kernel* [72]. When IBM introduced full virtualization to System/360, the result was a *hypervisor* that enabled software virtual machines and essentially supervised multiple supervisors [73]. The stronger prefix *hyper-* was chosen to imply a larger scope of authority than that of *super-*.

As stated earlier, the goal of this work is to move the point of isolation as close to the resource(s) as possible, thus *reducing* the size of the virtualization and preserving the authority of the kernel. In essence, this work proposes to introduce the *opposite* of a hypervisor. Just as hypothyroidism and hyperthyroidism are opposites, the term *hypovisor* is used in this work to refer to a virtual object manager with a very limited scope of authority compared to that of a hypervisor. Where hypervisors have authority over one or more virtual machines and their kernels, hypovisors have authority only over specific objects within the operating system.

The term hypovisor is also used in [74], where LXCs were used as a hypovisor to allow multiple, separate, independent instances of the Android Framework to run simultaneously on the same hardware.

5.3.2 High-level design overview

Figure 5.2 depicts the overall concept at a high level. Given a specific security goal that relates to App 2's interaction with objects A and B, our goal is to place the point of isolation (i.e., the

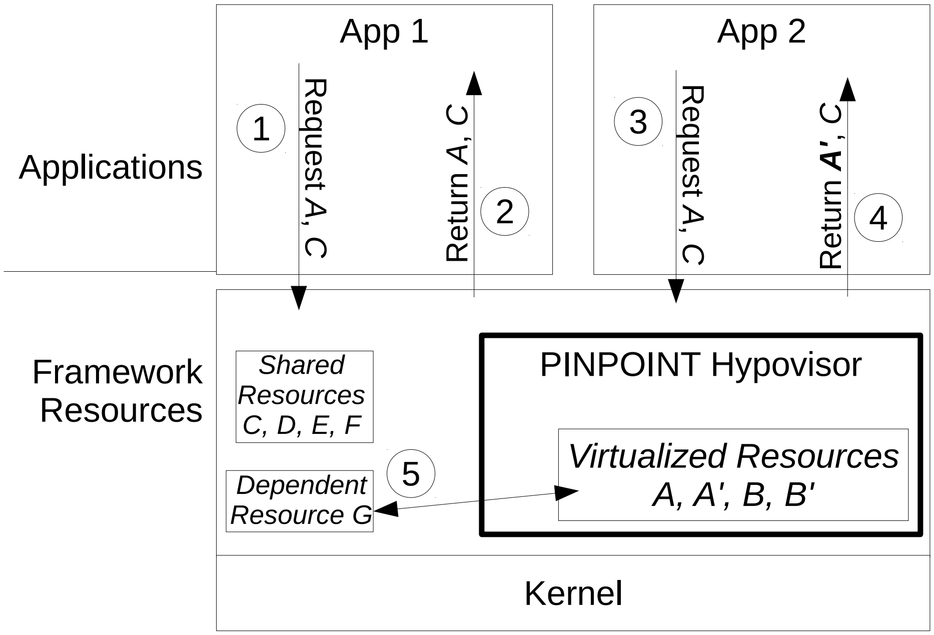


Figure 5.2: PINPOINT concept showing minimized isolation to address security goals, with maximized sharing of system objects.

hypovisor) at a strategic location that enables virtualization of only these objects such that App 1 and App 2 see different instances of each, while everything else about the system is common and unmodified. When trusted App 1 requests *A* and *C* (①), the hypovisor returns instances of *A* and *C* (②). On the other hand, when untrusted App 2 presents the same request (③), the hypovisor returns instances of *A*′ and *C* (④). Since *C* and other resources *D*, *E*, *F* and *G* are not related to the security goal, they are not virtualized, and either app may share them. Thus, the isolation size is minimized to just *A* and *B* according to the threat and stated requirements. Resources that can and must be shared for transparent operation remain shared as intended. In this way, Framework complexities that would arise from utilizing kernel-level isolation mechanisms are completely avoided.

A non-trivial challenge that can sometimes arise when PINPOINTing certain resources is when there are other operating system components or resources, shown in Figure 5.2 as resource *G*, that depend on interactions (⑤) with *A* or *B* and are unaware that now multiple virtual copies of them exist. In these cases, G must also be modified to account for this. Since this represents additional complexity, one must always consider whether this extra complexity will negate the lightweight benefits of the PINPOINT approach. If A is a large and complex object that has many dependencies throughout the system, it's likely that creating virtual copies of A will break many things that assume there is only one A. In cases like these, it may be better to use a coarser isolation such as the approaches in previous works. On the other hand, if A has few dependencies, then the modifications to G (if in fact there are any) will be straightforward. Two of our four case study applications described in Section 5.5 exhibit this characteristic, and these details are included there.

Another challenge that can arise is when the same sensitive information can be revealed by more than one object. For example, let's say both A and C are capable of returning a piece of sensitive data such as IMEI. It is important that all of these paths be identified, and either blocked or added to the isolation boundary. Our case study encountered one example of this which will be described in Section 5.5.

When designing and implementing the hypovisor, care must be taken to ensure that the system does not allow any form of delegation of the hypovisor's duties. For example, if the hypovisor is responsible for dispatching a capability, there must be no other ways for an entity to acquire that capability. Any other ways must be blocked in order to maintain the integrity of the isolation. Section 5.4.3 contains a specific example of this and how we addressed it using mandatory access controls (MAC).

5.3.3 Methodology

A summary of the PINPOINT methodology is found in Table 5.3.

Identifying the best place to instantiate the hypovisor is key to achieving a balance between flexibility and specificity. In our experiences thus far, we have found that the best isolation points

Table 5.3: PINPOINT Methodology.

Step	Description	Example
1	Define/collect security goal(s)	Protect IMEI from app A
2	Identify relevant resource(s)	*iphonesubinfo* and *phone* system services
3	Identify point(s) of resource access / capability dispatch → implement hypovisor(s) & generalize	**`servicemanager`**
3a	Security analysis	Prevent inter-app passing of service binder tokens (modify MAC policy)
4	Identify and address dependency(ies)	**`com.android.phone`** and *ProxyController* (service startup)

are places in the Framework where classes of objects and/or their capabilities are managed or dispatched to apps. In Android, many resources are abstracted as system services, and their capabilities are dispatched by *ContextManager* (i.e., the native **`servicemanager`** process). As such, our initial work has focused on PINPOINTing system services, and our accomplishments thus far in this regard are described in Sections 5.4 and 5.5. However, we see future opportunities for implementing complementary hypovisors in key places other than system services, including:

1. High level data objects, such as *ContentProvider*. These may leak personal data [75].
2. Binder and Intents. These may be used as a path for a malicious app to attack or trick other apps[76].
3. Camera, audio. These have obvious privacy implications if miused, e.g., [21].
4. Clipboard. Can be used as an attack channel [77].
5. Accessibility subsystem. May be malicious toward critical apps [78].
6. Notifications. Potential misuse [79].

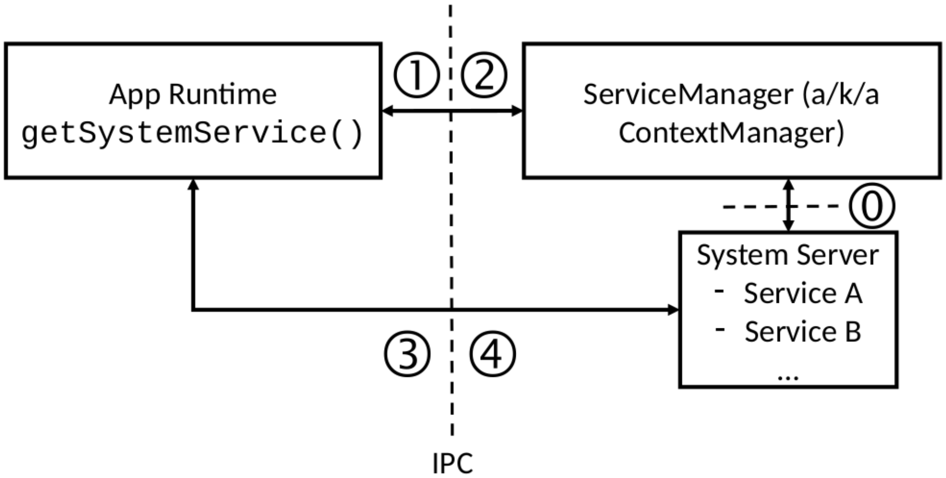


Figure 5.3: Interactions with System Services.

5.4 Case Study on Android System Service

In order to evaluate the PINPOINT concept, we undertook a case study using Android's system services framework in both Android 4.4.4 (KitKat) and 5.1 (Lollipop) on a Nexus 5 device. Since a wide variety of key resources are abstracted as system services, this choice illustrates that if the point of isolation is wisely chosen, a single PINPOINT hypovisor can be used for a variety of situations. To illustrate this point, we first provide some background on system services.

5.4.1 Android System Services

Interactions between Android applications and system services is enabled by the *Binder* and *ServiceManager* subsystems. Binder relies on capability-based security and implements a "call by invitation" mechanism to allow communication among apps, system services and *ServiceManager*. As such, before an app is allowed to call a service, it must receive an invitation in the form of an `IBinder` token.

Figure 5.3 shows an overview of the process by which apps access resources presented as system services. Invitations are first created when services are registered with the central directory of services known as *ContextManager*. By design, there can be only one *ContextManager*, a designation granted exclusively to the native `servicemanager` process early during the boot process, by way of its privileged relationship with *Binder*.[2] Once *ServiceManager* becomes *ContextManager*, *SystemServer* registers core system services using the `addService()` method of *ServiceManager* (path ⓪). The result of this registration process is that *ServiceManager* now holds an invitation (`IBinder`) for every system service running on the device. When an app needs an invitation for one of these services, it contacts *ContextManager* (path ①). *ContextManager* then passes a copy of this invitation to the app (path ②), and upon seeing this transaction, *Binder* updates its protected list of invitations held by the app. Invitations cannot be forged because any forged invitation will not have a corresponding entry in the protected list maintained by *Binder*. Once the app has the invitation, it can interact directly with the service running in *SystemServer* (paths ③ and ④).

All requests for system services, even those made by system components, must go through *ContextManager*. Thanks to *Binder*, the native `servicemanager` process has access to the trusted identity of the caller, in the form of the Linux *uid*, which corresponds to the Android *userId* and *appId*. This makes `servicemanager` an excellent place to implement a system service hypovisor that can regulate applications' interactions with virtualized system services. In this way, this hypovisor represents the PINPOINT "sweet spot" of being specific enough to limit inter-dependencies with other parts of the system, but flexible enough to apply to a large class of objects and the mechanism whereby their capabilities are dispatched.

5.4.2 PINPOINTing System Services

The case study implementation is presented in three parts as shown in Figure 5.4, beginning with the central enabling core, the system service hypovisor, labeled ①. The hypovisor exists within the

[2] In fact, in the Android source (`frameworks/native/cmds/servicemanager/binder.h`), *ServiceManager* is described as "The One Magic Handle".

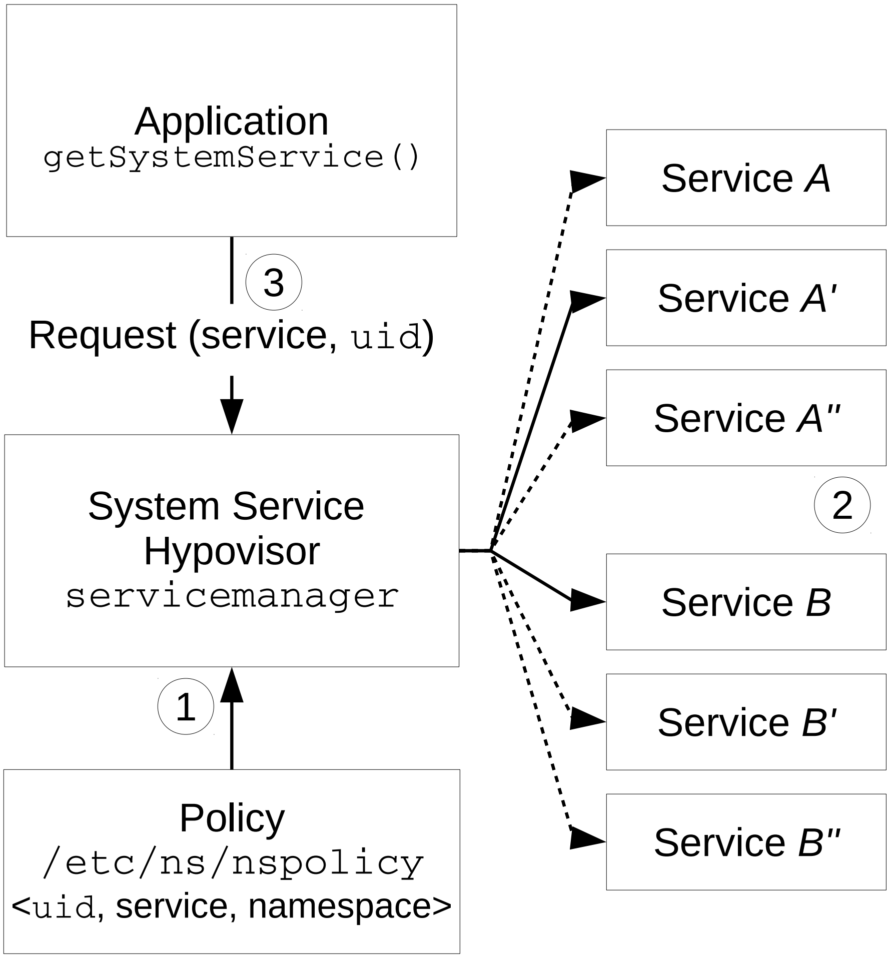


Figure 5.4: Design overview showing the service hypovisor and policy definition ①, virtual service plug-ins ②, and application ③.

native **`servicemanager`** process, where all service lookups are processed and capabilities dispatched. Lookup requests by apps (③), are initiated with a call to **`Context.getSystemService()`**, and arrive in the form of a *Binder* transaction containing the name of the service requested (e.g., **`location`** and other values as defined in **`Context`** class). Because the requests are *Binder* IPC transactions, they are identified by the app's Linux *uid* and *pid*. This identification can be trusted because it is applied in the kernel driver.

The native **`servicemanager`** looks up and returns the capability (handle) for the requested service to the caller in the **`do_find_service()`** function. This function uses the service string, along with the caller's *uid* and *pid* to decide whether the caller is allowed to receive an invitation to call the service. First, the device's mandatory access control (MAC) policy may prevent the calling process from receiving the handle, based on the subject's and object's SELinux security context. Second, the service itself may disallow the dispatch of its handle to isolated apps (i.e., those with *uid*s in the range $[AID\_ISOLATED\_START, AID\_ISOLATED\_END]$). The PINPOINT hypovisor is placed within this function, following the *pid* security context check, and prior to the *uid* range check. Refer to Listing K.1 in Appendix K during the following description of the modified `do_find_service()` function.

Following the MAC policy check, the modified code consults a secure namespace policy file, `nspolicy`, using the requester's *uid* as the index. Namespace policy is defined by a set of 3-tuples of $< uid, service\_name, namespace >$, where *uid* corresponds to the *appId* of the app assigned to *service_name* namespace *namespace*. If the *uid* key appears in the policy, then `servicemanager` invokes the local function `add_ns()` (see Listing K.2) to concatenate *_namespace* to the *service_name* string that was passed to the function. A pointer to the string, along with a length value are passed to the function `find_svc` where the actual handle is retrieved. For *uid*s that do not appear in the policy file, `add_ns()` is not called and the original requested string is at the pointer location. Either way, `find_svc()` returns the actual handle of the service corresponding to the resulting modified or unmodified string. The policy may specify more than one *service_name* and *namespace* for a given *uid* to contain apps that present a multi-dimensional threat. Since handle lookup requests can occur once or many times during the lifecyle of an app, the design also supports dynamic policy changes.

An example policy file is shown in Listing 5.1. In this example, apps represented by *uid* 10052 and 10065 are assigned to *location* namespace 1, while app 10063 is assigned to *location* namespace 2 and *iphonesubinfo* namespace 1. 10064 is also assigned to *iphonesubinfo* namespace 1. Finally, as directed by the special *uid* value 99999, *all* apps are assigned to *sensorservice* namespace 1. Note that the *uid* can be chosen to address security needs related to multiple device users, such as the multi-user framework issues identified in Chapter 3. For example, to assign app 10063 to an alternate location namespace *only* for *userId* 10, `nspolicy` would contain the line `1010063 location 2`.

Listing 5.1: Example `/etc/ns/nspolicy` showing how apps are assigned to different namespaces (for the device Owner, `userId` 0).

```
10063 location 2
10063 iphonesubinfo 1
10064 iphonesubinfo 1
10065 location 1
10052 location 1
99999 sensorservice 1
```

Currently, virtual services shown in Figure 5.4 at ② (e.g., A', A'', B' and B'') are preconfigured at build time. For example, the existing *SensorService* code can be duplicated as *SensorService_1* and *SensorService_2*. The semantics of these duplicates is then modified such that they implement the required characteristics of the alternate namespaces, such as random data, "fuzzed" data, or adjustments to specific sensors or sensor families. These additional services are started at boot time, along with the normal global service. *SystemServer* is modified to start the additional services along with their global counterparts, A and B. An example of how *SystemServer* was modified to accomplish this for *LocationManagerService* and two alternate location namespaces is shown in the code excerpt of Listing 5.2. In this example, the running device will have two new services available, `location_1` and `location_2`, in addition to the original service, `location`. Listing 5.3 is an excerpt from Android's `service list` command, showing services with multiple namespaces for iphonesubinfo, location, input_method, and sensorservice running on the test device.

Listing 5.2: SystemServer excerpt showing how multiple instances of LocationManagerService are started.

```
public final class SystemServer {
  ...
  LocationManagerService location = null;
  LocationManagerService_1 location_1 = null;
  LocationManagerService_2 location_2 = null;
  ...
  if (!disableLocation) {
      try {
          Slog.i(TAG, "Location Manager");
          location = new LocationManagerService(context);
          ServiceManager.addService(Context.LOCATION_SERVICE, location);
      } catch (Throwable e) {
          reportWtf("starting Location Manager", e);
      }

      // register additional location namespace (1)
      try {
          Slog.i(TAG, "Location Manager 1");
          location_1 = new LocationManagerService_1(context);
          ServiceManager.addService("location_1", location_1);
      } catch (Throwable e) {
          reportWtf("starting Location Manager 1", e);
      }

    // register additional location namespace (2)
      try {
          Slog.i(TAG, "Location Manager 2");
          location_2 = new LocationManagerService_2(context);
          ServiceManager.addService("location_2", location_2);
      } catch (Throwable e) {
          reportWtf("starting Location Manager 2", e);
      }
      ...
}
```

Listing 5.3: List of services running on a Nexus 5 device, showing multiple namespaces for iphonesubinfo, location, input_method, and sensorservice.

```
$ adb shell service list
Found 105 services:
0   sip: [android.net.sip.ISipService]
1   phone: [com.android.internal.telephony.ITelephony]
2   isms: [com.android.internal.telephony.ISms]
3   iphonesubinfo_1: [com.android.internal.telephony.IPhoneSubInfo]
4   iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
...
36  location_2: [android.location.ILocationManager]
37  location_1: [android.location.ILocationManager]
38  location: [android.location.ILocationManager]
...
62  input_method_1: [com.android.internal.view.IInputMethodManager]
63  input_method: [com.android.internal.view.IInputMethodManager]
...
91  batterystats: [com.android.internal.app.IBatteryStats]
92  sensorservice_4: [android.gui.SensorServer]
93  sensorservice_3: [android.gui.SensorServer]
94  sensorservice_2: [android.gui.SensorServer]
95  sensorservice_1: [android.gui.SensorServer]
96  sensorservice: [android.gui.SensorServer]
...
104 drm.drmManager: [drm.IDrmManagerService]
```

Although not required, the alternate services usually have interfaces identical to their global counterpart, and differ only in semantics. For example, the global location service returns the actual current location, while the other location services return noisy, random or preset locations via an identical public interface. This example is illustrated by Listing 5.4 where latitude and longitude data received from the provider are overwritten with random values before being passed to the handler.

Listing 5.4: Modified ***`reportLocation()`*** *excerpt from LocationManagerService_1 showing how the API's semantics are changed from that of the standard LocationManagerService. In this case, the method overwrites the Location object with random lat/long values.*

```
public class LocationManagerService_1 extends ILocationManager.Stub {
    ...
    @Override
    public void reportLocation(Location location, boolean passive) {
        // This is the random location namespace, so overwrite real location that came from provider
        double lat = Math.random()*181.0-90.0; // -90 to +90
          double lon = Math.random()*361.0-180.0; // -180 to +180
          Log.d(TAG, "reportLocation() randomizing lat/lon to: "+ lat + "/" + lon);
          location.setLatitude(lat);
          location.setLongitude(lon);
          ...
          mLocationHandler.removeMessages(MSG_LOCATION_CHANGED, location);
        Message m = Message.obtain(mLocationHandler, MSG_LOCATION_CHANGED, location);
        m.arg1 = (passive ? 1 : 0);
        mLocationHandler.sendMessageAtFrontOfQueue(m);
      }
      ...
}
```

As mentioned in Section 5.3.2, some services may have dependencies outside the PINPOINTed virtualization boundary. An example of this is *LocationService* which receives "push" updates in the form of callbacks from native code. The implication of this is that the alternate location namespaces will not receive callbacks from the native code, since the native code was not designed with multiple location service instances in mind. Thus, this native code must be modified to support the additional services, resulting in additional complexity. In the case of location service, this additional complexity is relatively small, and is implemented by changing the scalar callback object to an array of callback objects with length equal to the number of location namespaces. Listing 5.5 shows how this was done for the native function `location_callback()` in the GPS provider.

Listing 5.5: ***location_callback()*** *from native GPS provider modified to push location updates to all registered location services by way of an array of callback objects.*

```
static void location_callback(GpsLocation* location)
{
    JNIEnv* env = AndroidRuntime::getJNIEnv();
    //env->CallVoidMethod(mCallbacksObj, method_reportLocation, location->flags,
    for (int i = 0; i < 3; i++) {
         if (mCallbacksObj[i]) {
            env->CallVoidMethod(mCallbacksObj[i], method_reportLocation, location->flags,
               (jdouble)location->latitude, (jdouble)location->longitude,
               (jdouble)location->altitude,
               (jfloat)location->speed, (jfloat)location->bearing,
               (jfloat)location->accuracy, (jlong)location->timestamp);
            checkAndClearExceptionFromCallback(env, __FUNCTION__);
         }
    }
    //
```

5.4.3 Security Discussion

The PINPOINT case study introduces a lightweight services hypovisor into the native portion of *ServiceManager*. The purpose of the hypovisor is to isolate particular apps from various services as specified by the user's policy. This security discussion is included to provide a sense of the strength of this isolation. We begin with *Binder*, since most of the isolation strength derives from *Binder*'s security model.

Every process using *Binder*, including system service threads within **`system_server`** has a protected representation in the kernel as an instance of a **`binder_proc`** structure. Each remote capability that a process holds is represented by one or more **`binder_node`** structures attached to the **`binder_proc`** instance. These nodes are known only to the kernel module and are used to determine the recipient of the communication, based on a handle provided from userspace. Handles are local references and mappings from handles to nodes are also stored securely in **`binder_proc`**. Hence, only the kernel knows how to map a particular handle to the corresponding node.

When *SystemServer* registers system services with *ContextManager* using **`addService()`** (as shown in the example of Listing 5.2), the kernel adds the service's `binder_node` to the `binder_proc` corresponding to the `servicemanager` process. *ContextManager* also allocates a local index to each registered service. When an app asks *ContextManager* for a handle to a service, `servicemanager` returns the handle and the kernel binder driver adds the service's `binder_node` to the app's `binder_proc`, thus recording the validity of the app's newly acquired capability. Apps can also send handles they posses to other apps via Intent. Upon seeing the handle within the transaction, the kernel driver adds the node to the receiver's `binder_proc` so that the recipient is now a valid holder of that capability. This is known as a *binder transfer*.

The addition of a services hypovisor does not change anything about *how* handles are looked up and provided by *ContextManager* or *how* capabilities are propagated by way of the kernel binder driver. All apps, native and Java alike, are subject to the intervention of the hypovisor when requesting service handles from *ContextManager*. Thus, any vulnerabilities in the prototype regarding *how* service handles are obtained from *ContextManager*, or vulnerabilities in the binder driver itself, are also vulnerabilities of stock Android and thus outside the scope of this discussion.

What the design does change is *which* handles are given out. PINPOINTing services introduces the notion of remote service handles that should be unobtainable by certain apps. For example, an app assigned to an alternate location namespace should *never* be allowed to get the capability to the global location namespace, either directly from *ServiceManager or from another app*. This is different than stock Android where *ContextManager* acts as an open directory service, and obtaining a service handle via binder transfer from another app does not represent a capability leak. In fact, the kernel binder driver itself facilitates these transfers through the `BINDER_READ_WRITE` command of the driver's `ioctl` interface. When a transaction is made, the driver simply updates the target's `binder_node` to reflect the newly-acquired capability. These app-to-app binder transactions do not involve `servicemanager` and therefore do not involve the PINPOINT services hypovisor. Thus, in order to maintain isolation among namespaces, these transfers must be regulated or blocked.

In essence, the hypovisor design turns an unprivileged instruction for transferring binder capabilities into a privileged one. Thus to enforce this new privilege, all ways to transfer binder capabilities must be controlled. Thankfully, all binder transactions, including transfers, are processed in one place, `binder_transaction()` function of the binder driver code, `binder.c`.

Binder's `ioctl` interface supports five commands: `BINDER_WRITE_READ`, `BINDER_SET_MAX_THREADS`, `BINDER_SET_CONTEXT_MGR`, `BINDER_THREAD_EXIT`, and `BINDER_VERSION`. Of these, only `BINDER_WRITE_READ` is of interest for the analysis of binder transfers, as it is the basis for all IPC operations. `BINDER_WRITE_READ`'s purpose is to copy a binder data buffer from userspace into the kernel for processing, and then back to userspace when processing is finished. The buffer includes a `binder_transaction_data` structure which contains the handle of the target, transaction code, and other information about the transaction including the sender's *pid* and effective *uid*.

The `BINDER_WRITE_READ` command is handled by a `switch` statement in function `binder_ioctl()` in `binder.c`. The `BINDER_WRITE_READ` case branches to two places, `binder_thread_write()` and `binder_thread_read()`. Binder transactions are ultimately accomplished through the former, so the focus now turns to `binder_thread_write()`.

`binder_thread_write()` does a lot of bookkeeping, but for binder transactions, it in turn relies on `binder_transaction()`. After validating the source and target of the requested transaction, the core of `binder_transaction()` includes a `switch` statement where each of the three possible types of binder objects are processed accordingly. These include binders, handles and file descriptors. Because they take the form of binders and handles, the focus of controlling transfers lies within these two parts of the `switch`. Inside the `case` blocks for binders and handles, the `binder_proc` and `binder_node` kernel data structures are updated according to the transaction. These are the only two places in the driver where this is done. Hence, we add an access control point in these two locations to enforce the new privilege associated with binder transfers.

Theoretical transfers of capabilities by means other than a binder `ioctl`, such as through UNIX sockets, will be ineffective as the kernel binder driver will not record the transaction and will not update the target's `binder_proc`. The received capability will be useless.

The PINPOINT system services prototype blocks app-to-app transfers of system services capabilities in the kernel binder driver's `binder_transaction()` function, through an extension of existing SEAndroid MAC policy. Specifically, the `security_binder_transfer_binder()` hook, present in each of the two relevant `case`s of the transaction type `switch`, is extended to also pass the `task_struct` of the `binder_ref` (for references) or `binder_node` (for handles) under consideration. This allows the hook function to extract the owner's SELinux security identifier (SID) and use it as the subject in an access control decision. To enforce the prohibition on app-to-app transfers of system service handles, the prototype contains modified type enforcement rules pertaining to `untrusted_app`s, so as to disallow transfer of `u:r:system_server:s0` binders between apps with an SELinux security context of`u:r:untrusted_app:s0`. In addition, a new `neverallow` rule was added to further ensure at policy build-time that there are no `allow` rules elsewhere in the policy that are inconsistent with this. This effectively blocks any attempted bypass of the hypovisor, while allowing all other normal binder transfers among apps and the system to proceed.

In assessing the impact of this new binder transfer restriction, a legitimate use of the function was sought out, with no findings. In fact, even after extensive testing of many apps, none that exploit this possibility have ever been found. Seemingly, the normal design patterns have apps getting system services directly from `servicemanager` rather than relying on other apps. Thus, there is high confidence that blocking these transfers will have no affect on any legitimate app.

5.4.4 Policy Configuration

As explained in Section 5.4.2, the `servicemanager` hypovisor consults a secure policy file to determine if the requester has been assigned to any alternate service namespaces. This policy can be created and updated by a variety of means: via the system Settings app, via launcher configuration, from hard-coded (i.e., build-time) mandatory policy, via over-the-air (OTA) updates in a mobile device management (MDM) architecture, via `adb`, using a privileged text editor on the device, etc. In the prototype, a default policy file was included in the system build, and it was updated via `adb` over USB and directly on the device using a text editor with root privileges. The system was also integrated with a custom launcher application being developed by another student. In terms of user-friendliness, the custom launcher enables the end-user to drag-and-drop app icons to and from different containers, each representing a specific PINPOINT configuration. For example, a particular container might be configured to protect two sensitive resources, location and IMEI, from the apps placed within it. When an app is dropped into this container, the launcher app automatically updates the policy with the `uid` and service names corresponding to the protected resources. This update takes effect immediately since `servicemanager` consults the policy each time the app makes a request.

5.4.5 Limitations

Currently, the case study prototype requires all global and virtual system services to be running whether or not any apps are assigned to them. In terms of overhead, this fact manifests itself as additional memory use by the `system_server` process. Although data presented in Section 5.6.1 shows that this overhead is small, this aspect of the design should be made more elegant and efficient in the future.

It is also important to note that the design does not provide full security domain isolation in the sense that it does not prevent apps from passing high-level sensitive information to other apps.

5.5 Applications

This section describes application and evaluation of the system services PINPOINT prototype in four practical applications involving system services. Each application involves a specific security goal that is used as the motivation for the scenario. All implementations were tested using AOSP branches of Android 4.4.4 (KitKat) and Android 5.1 (Lollipop) on a Nexus 5. The four services and corresponding security scenarios discussed here are as follows:

1. `LocationManagerService`: A widely used location-finding service that binds with a number of abstract provider mechanisms. Security goal is to prevent untrusted apps from obtaining accurate location information[80]. See Section 5.5.1.
2. `IPhoneSubInfo`: A "hidden" service for accessing phone subscriber information, called only by other system services such as `TelephonyManager`. Security goal is to prevent untrusted apps from accessing sensitive subscriber information [37]. See Section 5.5.2.
3. `InputMethodManagerService`: A service that arbitrates communications between apps and a variety of installed input methods, and has complex interactions with other system objects including `WindowManager`. Security goal is to protect critical apps from falling victim to malicious input methods[81]. See Section 5.5.3.
4. `SensorService`: A native service that interfaces directly with hardware sensors. Security goal is to prevent untrusted apps from obtaining accurate sensor data to steal data[23][82], eavesdrop [83], or track movement/location[84]. See Section 5.5.4.

By and large, porting from 4.4.4 to 5.1 was straightforward. In fact, the implementation of the hypovisor within `servicemanager` is virtually identical, even across the major version releases. Most of the difficulty in porting to different Android versions is due to changes in the design of the services themselves, especially for internal services that are not designed to be directly accessed by developers and are thus not subject to deprecation. For example, between Android versions 4 and 5, the underlying architecture of (`IPhoneSubInfo`) changes substantially, requiring an expansion of the isolation boundary in order to continue to meet the security goal. This is discussed below.

5.5.1 Location Service

Although location services provide great convenience and enable new functionality for users, they have significant security and privacy implications if misused. While some apps require accurate location to fulfill their main purpose, others utilize location information only to enrich their primary function. For example, a social networking app's primary function is to interact with friends via photo and status updates. These apps usually enrich this interaction by attaching location to these updates. If the end-user wishes to prevent only this one app from knowing location, and still enjoy its primary friend-interaction functions, she must rely on the trustworthiness of the app's own settings and controls. This is because current location privacy support from Android itself is too coarse-grained to achieve the user's goal of isolating only this one aspect of this one app. If the app is poorly-written or malicious in its handling of location data, privacy leaks may occur despite the user's best efforts to prevent them. By PINPOINTing the location service resource, and placing only this app in the new location namespace, we can transparently and effectively address this user's security goal without inconveniencing her or introducing the complex system modifications and overhead of general-purpose solutions.

To demonstrate this, we PINPOINTed the location service to provide three separate location namespaces for assigning apps, each with different semantics but identical interfaces. The *global location namespace* functions normally and is used with trusted apps. A *fuzzy location namespace* provides reduced-accuracy location information by adding noise to location objects. Finally, a *random location namespace* returns totally random location data to assigned apps.

We implemented these two additional location namespaces by adding two additional system services, `LocationManagerService_1` (LMS') and `LocationManagerService_2` (LMS''), as shown in Figure 5.5. These present the exact same API as the stock service, and thus are indistinguishable from the app's perspective.

Each location service binds to the standard set of common location providers such as `GpsLocationProvider` that interfaces through native code to actual hardware. However, as alluded to in Section 5.3, these providers represent dependent resources (G in Figure 5.2) that are designed based on an assumption of only one location service. Thus, these must also be modified slightly to make callbacks to all three location services. Otherwise, LMS' and LMS'' will never get location update callbacks since the providers are not otherwise aware of the virtualized services. Since the specifics of these modifications was provided as an example in Section 5.4.2, they are not repeated here. Instead, refer to Listings 5.2, 5.4, and 5.5.

The semantics of the additional services are as follows: `LocationManagerService_1` replaces location updates returned from the providers with random data, while `LocationManagerService_2` adds random offsets to the same. Since each namespace is indistinguishable from the global location namespace in, apps in alternate namespaces behave normally and process the virtual location data as if it were real.

Figure 5.6 shows screenshots of a popular fitness app, *RunKeeper*[3], that we used to demonstrate the isolated location namespaces. The version used for testing accesses *LocationManagerService* directly and not through a proxy (see discussion below). Figure 5.6a shows points collected during an activity while the app is assigned the noisy location namespace. Figure 5.6b shows the same app while assigned to the random location namespace. Note that in both cases, the app's display indicates "Good GPS", demonstrating the complete transparency of these namespaces to this unmodified app.

In some cases, applications may use Google Play Services to access resources such as location [85]. In fact, the latest version of the *RunKeeper* app used for testing this namespace now accesses location in this way. In these cases, handles for *LocationManagerService* are obtained by the Play

[3] https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro

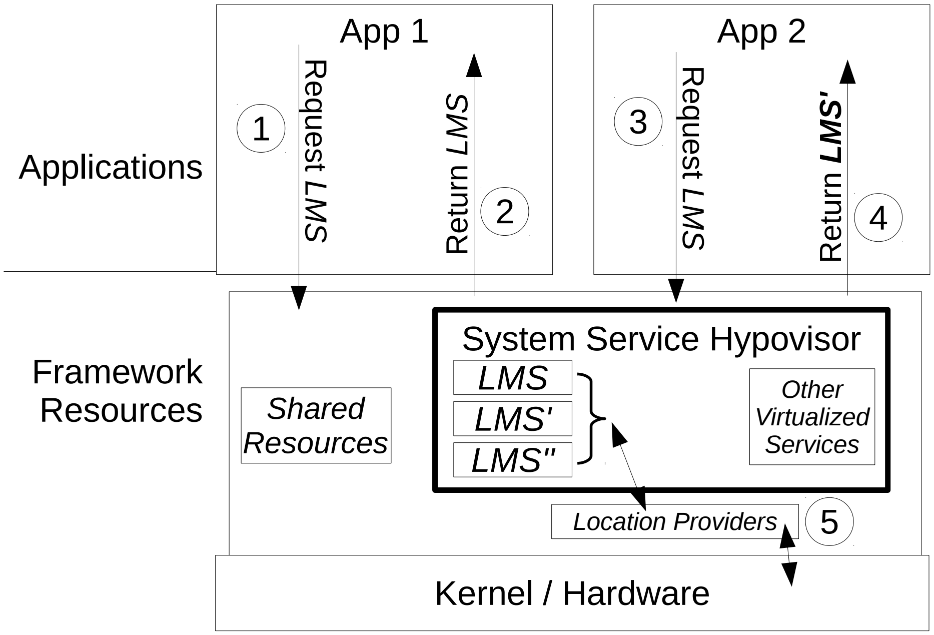
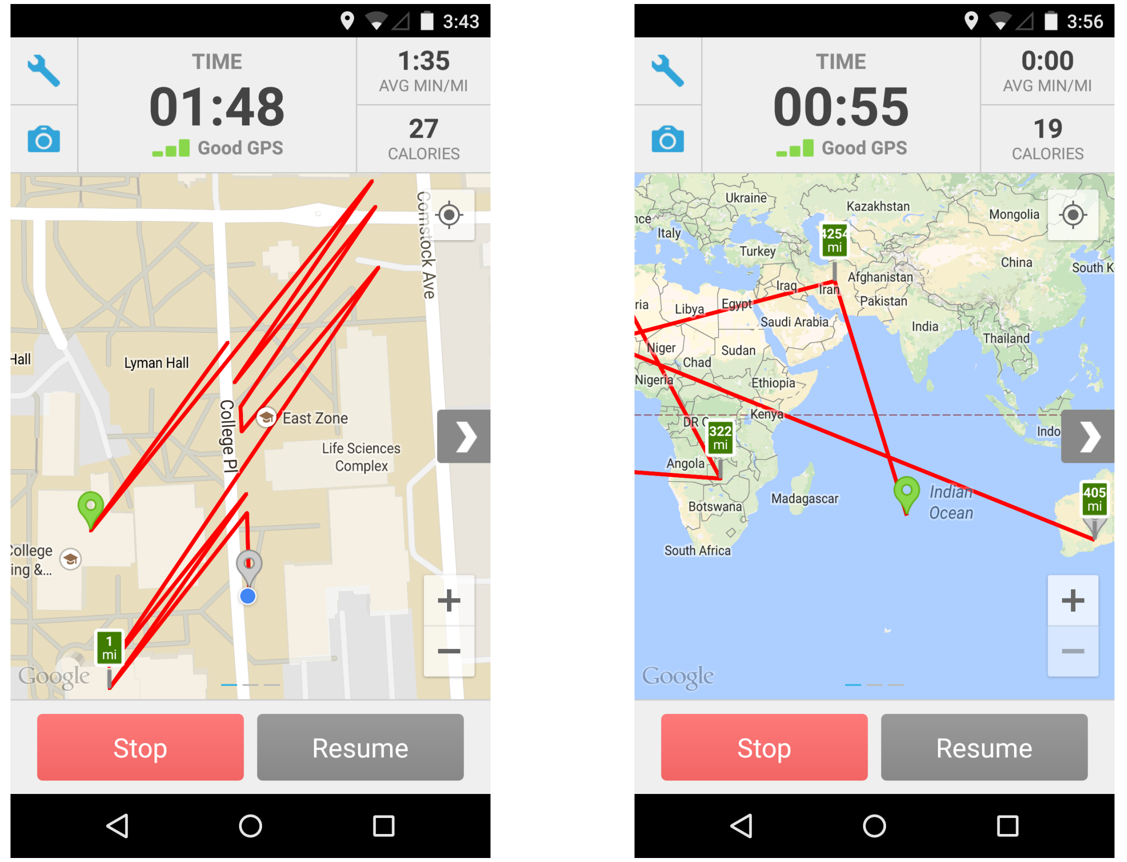


Figure 5.5: PINPOINTing `LocationManagerService`*.*

Services package, **`com.google.process.gapps`**, on behalf of the client app. Since this package runs as a separate app process, the system services hypovisor does not have access to the *uid* of the client. While the *uid* of the Play Services app can be assigned to a namespace, doing so will result in *all* apps using Play Services to be subject to the semantics of that namespace. While this is recognized as another type of dependent resource, G, addressing it is outside the scope of this work, since Play Services source code is not available. However, modifying Play Services to include an extension of the system services hypovisor would be possible.

5.5.2 Subscriber Information Service

`iphonesubinfo` is a hidden service used exclusively by **`TelephonyManager`** to service app requests for subscriber information such as IMEI, mobile equipment identifier (MEID), electronic serial number (ESN), phone number, voicemail number, private/public user identities, home network name, etc. Several of these values have significant security and privacy implications and are known to be malware targets [37]. Although protected by Android's **`READ_PHONE_STATE`**



(a) RunKeeper running in location namespace with noisy locations.

(b) RunKeeper running in location namespace with random locations.

Figure 5.6: RunKeeper fitness app running in alternate location namespaces.

permission, misusing or malicious apps can easily legitimize declaration of this permission since it is necessary for a number of common features, such as those provided by `PhoneStateListener`.

To isolate an untrusted app from or more of the data values returned by `iphonesubinfo`, we PINPOINTed this system service. We enabled the non-global namespace by modifying the internal telephony `ProxyController` to instantiate `PhoneSubInfoController_1` as well as `PhoneSubInfoController`. The former starts `iphonesubinfo_1` service with an API identical to `iphonesubinfo`, started by the latter. When an untrusted app is assigned to the alternate `iphonesubinfo` namespace, it can obtain the same instance of `TelephonyManager` as trusted apps can, but any subsequent calls to `getDeviceId()`, `getLine1Number()`, etc. by the untrusted app are processed by `iphonesubinfo_1`. `iphonesubinfo_1` returns different values for sensitive subscriber parameters.

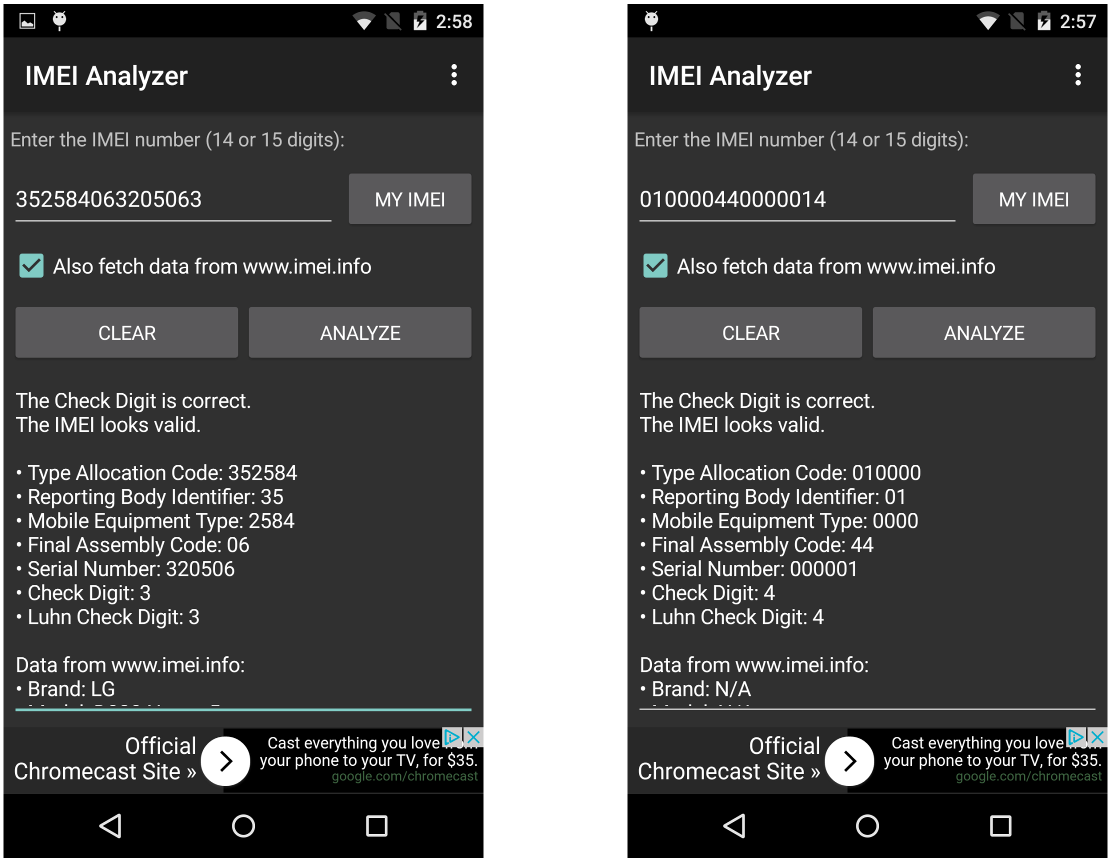
When porting this design to Android 5.1, we found that the underlying structure of the telephony service had changed significantly. In particular, the **ITelephony** (**phone** service) interface was enhanced to include its own **getDeviceId()** call, and **TelephonyManager** was modified to obtain the device ID from this interface rather than **IPhoneSubInfo** as was the case in 4.4.4. Thus, apps assigned to **iphonesubinfo_1** would still get the device's real IMEI because our isolation did not include every object that could return that sensitive data. This necessitates an expansion of the isolation boundary to include both **iphonesubinfo** and **phone** services, and is a good example of needing to identify all possible means of access to the sensitive resource related to the end security goal.

To demonstrate effectiveness of our PINPOINTed subscriber information service, we obtained the popular app *IMEI Analyzer*.[4] Figure 5.7 shows this app running unmodified in both global (Figure 5.7a) and fake (Figure 5.7b) **iphonesubinfo/phone** namespaces. In the global namespace, the actual, valid IMEI of our test device is returned, while a fake IMEI is returned to the app after it has been assigned to the alternate **iphonesubinfo_1/phone** namespace by adding its **uid** to the **nspolicy** file.

5.5.3 Input Method Service

Input Method Editors (IME) are screen controls that enable users to enter text. Currently, there are about 900 third-party keyboard apps published on the Google Play store, with at least 10 having more than one million downloads. Most require **INTERNET** or **WRITE_EXTERNAL_STORAGE** permissions, which enable the IME to log or transmit any data that's typed in. In an empirical study of keyboard apps, it was found that more than 61% require three or more permissions giving them the ability to exploit keylogging and man-in-the-middle attack vectors [81]. To illustrate this threat,

[4] https://play.google.com/store/apps/details?id=org.vndnguyen.imeianalyze



2:58

IMEI Analyzer

Enter the IMEI number (14 or 15 digits):

352584063205063

MY IMEI

☑ Also fetch data from www.imei.info

CLEAR

ANALYZE

The Check Digit is correct.
The IMEI looks valid.

- Type Allocation Code: 352584
- Reporting Body Identifier: 35
- Mobile Equipment Type: 2584
- Final Assembly Code: 06
- Serial Number: 320506
- Check Digit: 3
- Luhn Check Digit: 3

Data from www.imei.info:
- Brand: LG

(a) IMEI Analyzer running in global `iphonesubinfo/phone` namespace.

2:57

IMEI Analyzer

Enter the IMEI number (14 or 15 digits):

010000440000014

MY IMEI

☑ Also fetch data from www.imei.info

CLEAR

ANALYZE

The Check Digit is correct.
The IMEI looks valid.

- Type Allocation Code: 010000
- Reporting Body Identifier: 01
- Mobile Equipment Type: 0000
- Final Assembly Code: 44
- Serial Number: 000001
- Check Digit: 4
- Luhn Check Digit: 4

Data from www.imei.info:
- Brand: N/A

(b) IMEI Analyzer running in alternate `iphonesubinfo/phone` namespace.

Figure 5.7: IMEI Analyzer running in different `iphonesubinfo/phone` *namespaces.*

consider sensitive apps like banking or purchasing apps, which often require users to enter bank card numbers or passwords for authentication. All entry of these values is done via the current IME, selected by the user. If the IME is malicious, an attacker can easily collect these values [86].

The overall working architecture of IMEs is shown in Figure 5.8. In every application's context space, there exists an instance of `InputMethodManager` (path 1) which is used to communicate with a system-wide service, `InputMethodManagerService`. When an input field comes into focus, the app's `InputMethodManager` invokes this system service (paths 4 and 5) after obtaining its handle via *ServiceManager* (paths 2 and 3). With this handle, the app may obtain a unique `InputConnection` *Binder* token from `InputMethodManagerService` for making direct calls to the IME keyboard app. Using this token, the system is able to secure and control interactions among multiple applications and multiple IMEs [87].

Currently, apps do not have control over the IME selected by the user. Instead, the system will bring up the user's selected IME whenever any text field comes into focus. While Google has recognized the security and privacy issues associated with this design [88], the current measures rely on the user to make wise choices regarding IME installation and selection. Using session information attached to each window instance by `WindowManager`, the Input Method Framework (IMF) ensures that only the active activity can get access to the data being entered. Furthermore, `InputMethodManagerService` ensures that all messages received from running IME applications are from the current user. Importantly, this includes messages for changing IMEs (i.e., messages resulting from calls to `InputMethodManager.setInputMethod()`), which are guarded with the token to ensure that they originated from explicit user selection. However, none of these protections will help if the IME itself is malicious or compromised and the user selects it.

PINPOINTing the `InputMethodManagerService` provides an effective mechanism to shield sensitive apps from falling victim to a malicious IME selected by a tricked user. Figure 5.9 shows the PINPOINT concept applied to input methods. This is accomplished by using the our PINPOINT service hypovisor prototype to virtualize `InputMethodManagerService`. In the figure, *IMMS* corresponds to the "real" `InputMethodManagerService` (`input_service`), while *IMMS′* is a second service (`input_service_1`), with an identical interface and features except for the fact that it holds only a subset of all available IMEs.

As suggested in Section 5.3, there are additional complexities with virtualizing IMEs due to dependencies with other objects in the system. Because of interactions with `WindowManager` mentioned above, minor modifications to `WindowManager` are necessary so that it can be aware of the all the `InputMethodManagerService` namespaces running and push updates about the current activity to all of them. As with location service, this situation corresponds to dependent resource *G* in Figure 5.2. To enable independent `InputConnection` from each app's `InputMethodManager` instance to each service, we created a Java interface which all of the

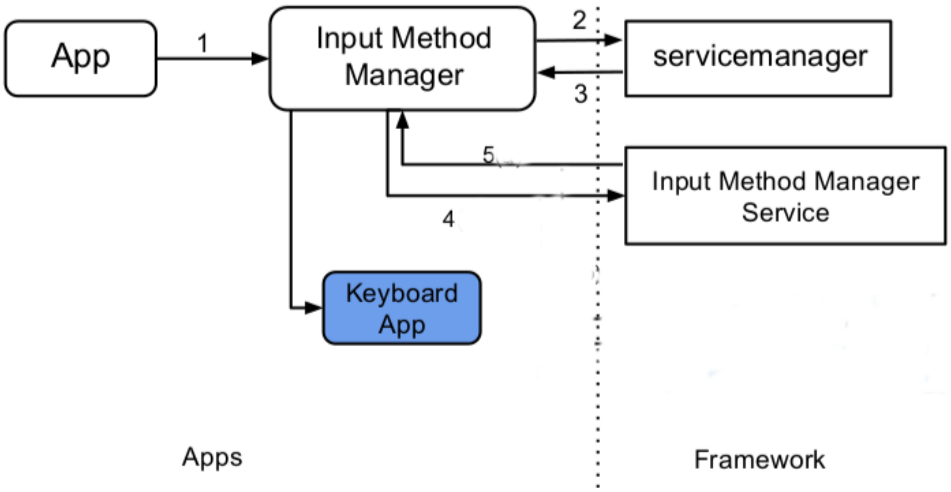


Figure 5.8: Input method framework architecture.

`InputMethodManagerService` instances implemented.

A demonstration of IME namespaces is illustrated by Figure 5.10. Figure 5.10a depicts a non-critical app, *Eat St.*,[5] assigned to the global IME namespace, where any IME can be used, including a representative untrusted IME, *SwiftKey*.[6] Here, the *Choose input method* dialog shows all installed input methods. In contrast, the critical banking app, Chase Mobile,[7] in Figure 5.10b has been assigned to the alternate IME namespace in order to protect its data from possible malicious IMEs. As shown, the chooser only allows selection of trusted IMEs, with *SwiftKey* excluded due to the alternate IME namespace isolation.

5.5.4 Sensor Service

Modern mobile devices have a rich set of environmental and motion sensors available to apps. Unfortunately, the Android security architecture does not extend to most of these sensors, making it all too easy for malware to utilize them to compromise user data entry [23,82], eavesdrop on voice communications [83], track user movements, and infer location [84]. By PINPOINTing

[5] https://play.google.com/store/apps/details?id=com.eatSt.app
[6] https://play.google.com/store/apps/details?id=com.touchtype.swiftkey
[7] https://play.google.com/store/apps/details?id=com.chase.sig.android

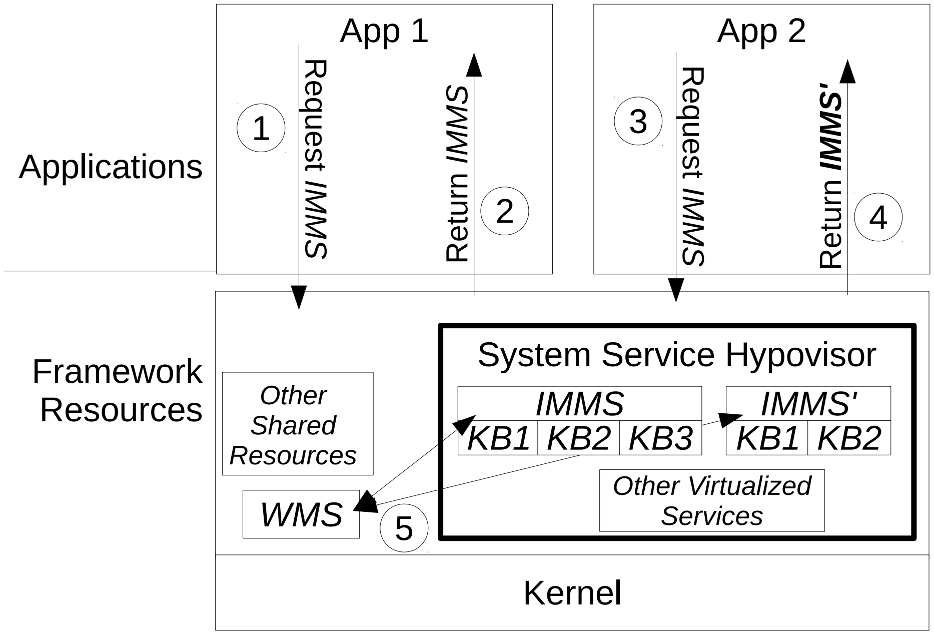
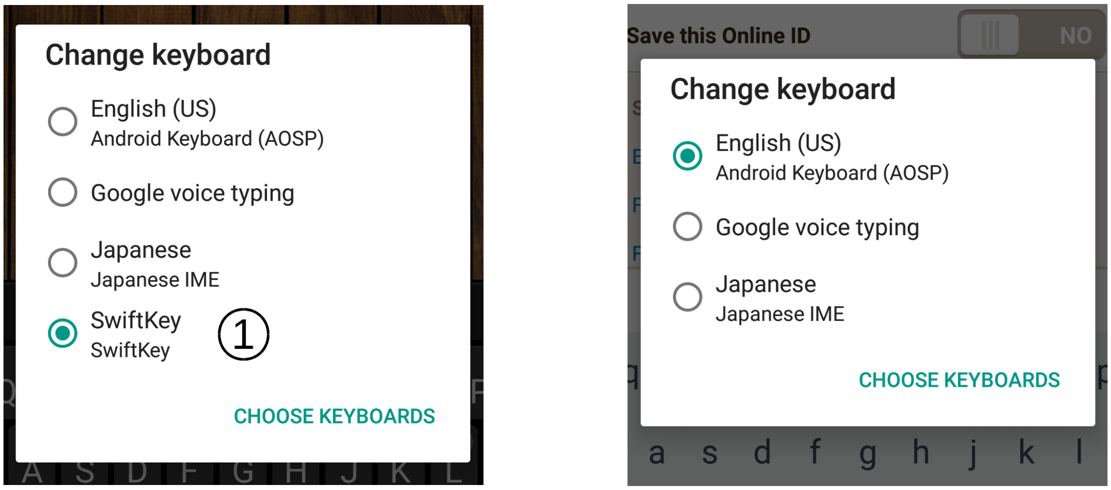


Figure 5.9: PINPOINTing `InputMethodManagerService`.



Change keyboard

○ English (US)
Android Keyboard (AOSP)

○ Google voice typing

○ Japanese
Japanese IME

● SwiftKey
SwiftKey ①

CHOOSE KEYBOARDS

(a) Non-critical app running in global IME namespace, showing all input methods, including a 3^rd^ party (①), as selection options.

Save this Online ID NO

Change keyboard

● English (US)
Android Keyboard (AOSP)

○ Google voice typing

○ Japanese
Japanese IME

CHOOSE KEYBOARDS

(b) Critical banking app running in alternate IME namespace, showing only built-in input methods as selection options.

Figure 5.10: Non-critical and critical apps running in different IME namespaces.

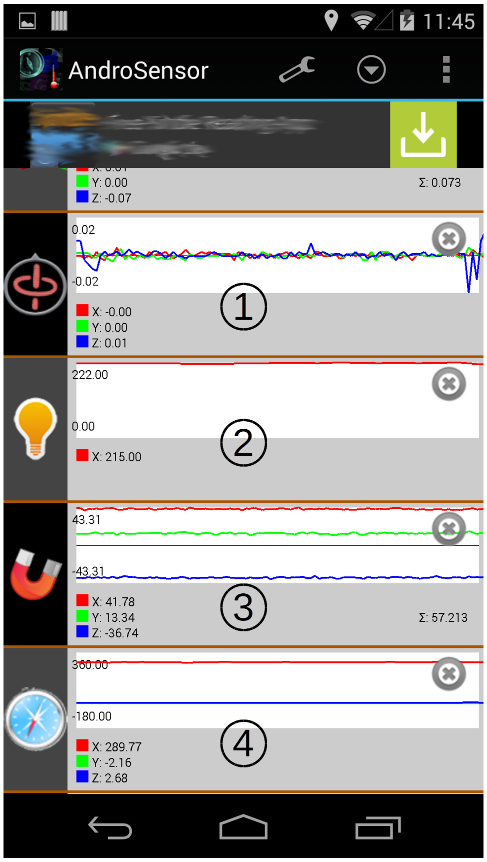
`SensorService`, we enable the user to take advantage of apps without needing to also trust their handling of sensor data.

In the Android platform, apps may acquire sensor data by getting an instance of *SensorManager*, which in turn accesses raw sensor data via *SensorService*, a native system service. *SensorService*'s **`threadLoop()`** collects raw sensor data in a structured data buffer of type **`sensor_event_t`**, which is then returned to the app via its *SensorManager*'s *SensorEventConnection*. The buffer structure contains raw sensor data for each of the device's sensors including acceleration, magnetic, orientation, gyro, temperature, distance, light, pressure, and relative humidity.

To PINPOINT sensor resources, we followed the same general approach as with previous examples, by adding two additional native *SensorService*s to the device, and registering them with *ContextManager* as **`sensorservice_1`** and **`sensorservice_2`**. For demonstration purposes, we hardcoded **`sensorservice_1`** to overwrite the gyro, magnetic, and orientation structure members of the buffer structure with random data before it is returned to the app's `SensorManager`. Likewise, **`sensorservice_2`** is hardcoded to overwrite only the light structure member of the structure with random values. Structure members containing data from other sensors are passed through unmodified.

With three possible sensor service handles on the device, `SensorManager`s of apps assigned to one of the two alternate sensor namespaces are always given handles to **`sensorservice_1`** or **`sensorservice_2`**, depending on their assignment. To demonstrate the effectiveness of this, we installed *AndroSensor*,[8] a popular Google Play Store app, and ran it in each of the three sensor namespaces. Figure 5.11 shows *AndroSensor* running in the global sensor namespace, with all sensor traces steady, indicating a stable physical environment. In contrast, Figures 5.12a and 5.12b show *AndroSensor* running in the alternate sensor namespaces of **`sensorservice_1`** and **`sensorservice_2`**, respectively. For all three cases, the physical envronment was approximately the same.

[8] https://play.google.com/store/apps/details?id=com.fivasim.androsensor



🟩 Y: 0.00
🟦 Z: -0.07
Σ: 0.073

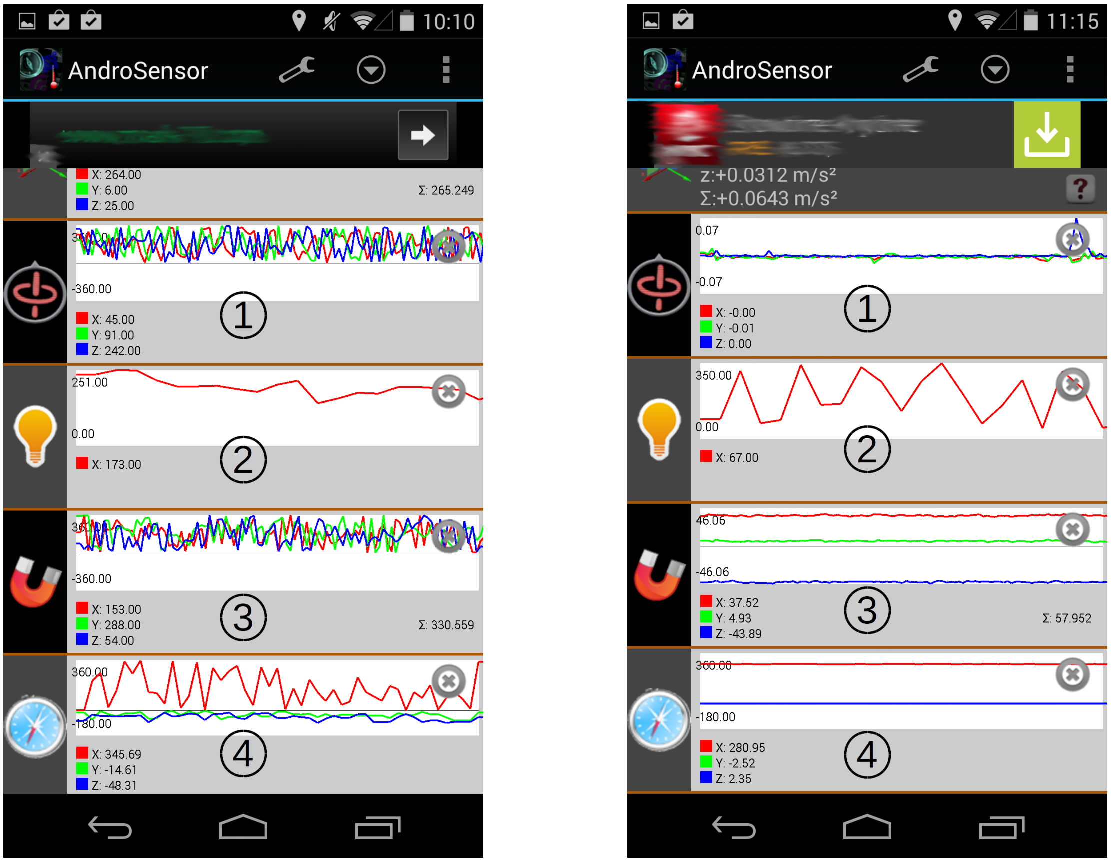
🟥 X: -0.00
🟩 Y: 0.00
🟦 Z: 0.01

🟥 X: 215.00

🟥 X: 41.78
🟩 Y: 13.34
🟦 Z: -36.74
Σ: 57.213

🟥 X: 289.77
🟩 Y: -2.16
🟦 Z: 2.68

Figure 5.11: AndroSensor running in global sensor namespace showing normal traces for gyro (①), light (②), magnetic (③) and orientation (④) sensors.



(a) AndroSensor running in 1st alternative sensor namespace showing normal trace for light sensor (②), and random traces for gyro (①), magnetic (③) and orientation (④) sensors.

(b) AndroSensor running in 2nd alternative sensor namespace showing normal traces for gyro (①), magnetic (③), and orientation (④) sensors, and random trace for light sensor (②).

Figure 5.12: AndroSensor running in alternative sensor namespaces.

Table 5.4: Evaluation benchmarks used.

Name	Version	Workload type
Linpack	1.2.8	CPU
Quandrant Advanced Edition	2.1.1	File I/O
Quandrant Advanced Edition	2.1.1	2D & 3D
SunSpider	1.0.2	CPU & I/O

5.6 Evaluation

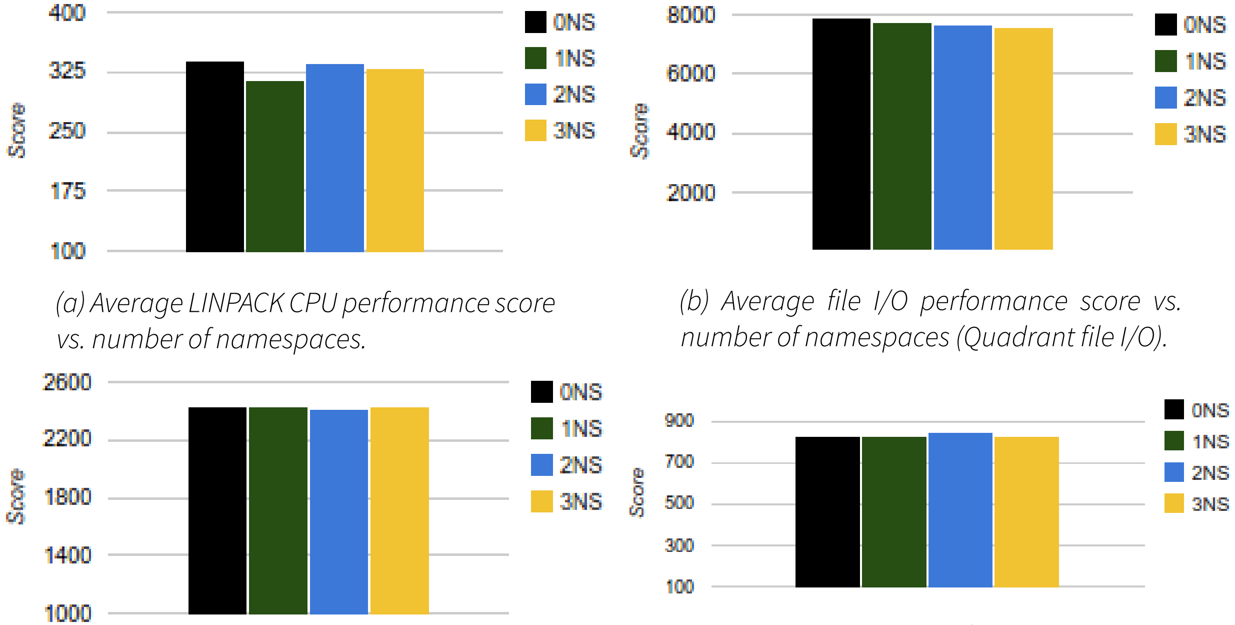
5.6.1 Performance

Evaluation of the overall performance impact of PINPOINTing services was accomplished with the benchmarking tests shown in Table 5.4, with and without namespaces. For each benchmark, performance under four different device configurations was measured: *0NS* represents stock Android without any PINPOINT capability or namespaces, while *1NS*, *2NS*, and *3NS* represent devices configured with one, two and three PINPOINTed services, respectively. Figure 5.13 shows the average value of 10 runs of each benchmarking test.

The impact on memory of adding PINPOINTed services was also measured. Since each running service represents additional threads within *SystemServer*, we measured *VmSize* of the **`system_server`** process by reading its **`/proc/<pid>/status`** under each of the same four configurations. Figure 5.14 shows the average value of 10 measurements of memory footprint for each configuration.

5.6.2 Qualitative Assessment

Having described four representative PINPOINTed services, and measured overall system performance impacts, we now recall the qualitative metrics related to effectiveness and efficiency



(a) Average LINPACK CPU performance score vs. number of namespaces.

(b) Average file I/O performance score vs. number of namespaces (Quadrant file I/O).

(c) Average file I/O performance score vs. number of namespaces (Quadrant file I/O).

(d) Average browser performance score vs. number of namespaces (SunSpider).

Figure 5.13: Benchmarking results for 0-, 1-, 2- and 3-namespace configurations.

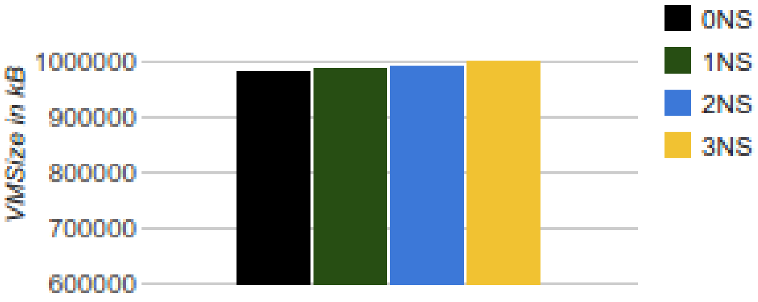


Figure 5.14: Average memory footprint in kB (VmSize) for 0-, 1-, 2- and 3-namespace configurations.

Table 5.5: PINPOINT prototype measured against qualitative metrics related to effectiveness and efficiency (L=low/small; M=medium/some; H=high/large).

Metric (best)		**PINPOINT**				**Comparative Approaches of [64] and [65]**
		Location	Sub. Info.	Input Method	Sensor	
Effectiveness	Range of threats (H)	L	L	L	L	H
	Adequacy (H)	H	H	H	H	H
	Isolation size (H)	L	L	L	L	H
Efficiency	Complexity (L)	L	L	L	L	H
	Transparency (H)	H	H	H	H	M
	Performance (H)	H	H	H	H	M
	Usability & convenience (H)	H	H	H	H	M

introduced in Section 5.4, and assess the prototype against them. Table 5.5 shows these results.

5.6.3 Discussion

Performance evaluation results presented in Section 5.6 indicate that increasing numbers of PINPOINTed services has a negligible effect on CPU, browser, and graphics performance. On the other hand, data indicates a clear correlation between the number of PINPOINTed services and file I/O. Decreases in this score with increasing numbers of namespaces is expected due to an increase in policy file size and associated data structures being parsed and searched by `servicemanager` during every service lookup request in order to support namespace reassignments of running apps. In our current, unoptimized design, file I/O performance degrades by an average of 1.57% of the *0NS* value for each additional namespace represented in the policy file. Although this degradation is negligible for simple policy files, we feel that this is an area for improvement. Future implementations will include an optimization of this code, and policy options to configure how often policy lookups are performed.

We also observed a growth in `system_server`'s memory size that is correlated to the the number of additional service objects (i.e., namespaces) available for use in the `system_server` process. On average, we observed this increase to be approximately 0.64% of the *0NS* value per each additional service. For a system with one additional IMEI namespace, two additional location namespaces, one additional input method namespace, and two additional sensor namespaces, `system_server` would have an approximately 3.84% larger memory footprint than the stock process. Note that an unused namespace still consumes additional memory, but since it does not add to the policy file, it will not contribute to file I/O degradation. This is another area target for improvement in the future.

5.7 Future Directions

In our present work, we have gained a tremendous insight into the trade-off between isolation design alternatives, system complexity, usability, convenience and effectiveness. We plan to further quantify these relationships so that we can make informed choices when addressing the high-level requirements typically stated by end-users. Ultimately, we plan to formalize the PINPOINT methodology, so that security designers can easily understand the trade space of PINPOINT designs vs. general-purpose approaches.

Through our case study of implementing a services hypovisor, we've acquired a sense for the difficulty of implementing a representative PINPOINT hypovisor and its companion virtual resources within the Android Framework. Encouraged by our experiences, we plan to consider the potential benefits of PINPOINTing other resources, including those outside the purview of *Service*

Manager. Following from this, we envision implementing a container abstraction, whereby multiple, heterogeneous PINPOINTs, Linux Namespaces, and other forms of access control and virtualization can be easily combined by the end-user to form easily-understood security and privacy macros such as “incognito,” “banking,” etc.

We recognize the inflexibility of having to define PINPOINTed resources at system build time. As such, we see opportunities to investigate techniques for establishing new PINPOINTs while the system is running. Also, we would like to evolve our current rudimentary means of policy configuration into a more powerful and intuitive means for end-users to configure, combine and use PINPOINTed resources, possibly through an advanced launcher interface.

Additional project details, status, and instructions for requesting access to our prototype code are available at `https://goo.gl/2pJeMp`.

Chapter 6

Related work

This chapter is organized as follows: Section 6.1 discusses previous work on security analysis techniques and specific types security and privacy problems in Android, and compares these with the contributions of Chapters 3 and 4. Section 6.2 compares the contributions of Chapter 5 with prior work on resource protection, sensitive data protection, and privacy.

6.1 Security Analysis

Work on systematic security analysis was initially inspired by those who have reported on confused deputies [89], [90], [91], [92], [93], [94], component hijacking [95], and capability leaks [41], [96] in Android. However, these are different from the work presented here for two reasons. First, their goal is to find and explain specific types of vulnerabilities rather than gaining an overall understanding of how access control is implemented along paths between specific sets of subjects and objects (Chapter 3), or within the modular resources themselves (Chapter 4). When subject-object paths are checked, both effective as well as missing or ineffective access controls are uncovered. These missing or ineffective controls may lead to one or more of the categories studied in the previous work, but new categories are also uncovered because of the deliberate choice to focus on subject-object combinations rather than a specific threat model.

Second, the contributions of Chapters 3 and 4 include unique case studies that result in findings different than previous work. The case study on the multiuser framework was described by a reviewer as the only known at the time of publication, and this still appears to be the case at the time of this writing. In fact, a fundamental difference is that that all of the previous work focuses on maliciousness among or by *apps*, while the multiuser work addresses potential maliciousness among *users*. Several studies [97,98], [99], [100] have explored the use of motion sensors available on smartphones to perform user activity recognition, while [22], [24], [23], [82], [25] have focused on inferring user keyboard presses, icon taps and secure inputs using accelerometer and gyroscope sensor. Nonetheless, these do not consider unauthorized use of these sensors by other users of the device as done here. Finally, [101] and [102] present complete secure multi-user architectures which may be able to solve some of the problems identified in Chapter 4.

Several works have contributed to finding security flaws and inconsistencies in Android. Kratos [47] uses static analysis to find access control inconsistencies in System Services. The authors define these as any condition where two entry points from the same image reach the same sink by different paths, and the security checks along the two paths differ. As such, Kratos cannot find *missing* access controls, or evaluate controls against special conditions such as user switching, as Chapter 3's methodology does. In addition, Kratos does not allow comparisons among different images, as is the case with the differential feature-based analysis of Chapter 4. Finally, Kratos depends on costly static analysis of the entire code path between entry point and resource, in contrast to the feature-based approach which favors rapid analysis as well as creation of a comprehensive feature database that can support a variety of research inquiries.

Identification and evaluation of custom system configurations, including the addition of receivers and custom permissions by device vendors is the subject of [46]. The authors extract features from system and app manifests and compare these to a baseline. Although similar in terms of the feature-based approach of Chapter 4, the actual feature sets are quite different, as are the feature extraction processes and portions of the system that are evaluated. Hence, the prior work and the work presented herein are complementary.

Problems with pre-loaded apps are addressed in several works [44,41,45]. As is the case with the custom configuration work just discussed, these evaluations focus on the apps and their configurations rather than the system code itself.

Hanging attributes are unused references that can be co-opted by malicious apps. [43] describes a means to identify these in custom system images. At a high level, this is essentially a special case of the feature extraction introduced in Chapter 4. However, the goal of Chapter 4 is to generate an overall characterization of access controls to support a variety of inquiries, while this work is specialized in identifying only hanging attributes. Nevertheless, the two contributions are complementary in that they each provide important assessments of security within the system itself.

The Linux kernel has been the subject of security analysis for many years. In 2000, [48] introduced an automated compiler-based method that finds system-specific rule violations in source code. Certain features of compiler error output are examined and matched against a set of rules designed to find security problems and other bugs. At a conceptual level, the features generated by the compiler extensions have several similarities to the features described in Section 4.2.1. Each are defined from domain knowledge through a feature engineering process. Each attempts to capture salient information about the system in an efficient albeit lossy representation. Finally, each is used to facilitate additional analyses necessary to make various conclusions about the system. However, in addition to the obvious differences between the Linux kernel and Android Framework, which make the feature sets entirely incompatible, the features of [48] are specific to finding particular bugs rather than providing a more general characterization of access controls as described in Chapter 4. In addition, the compiler-based approach requires source code, while the extraction process described in Section 4.2.2 is useful for proprietary vendor devices for which source code is not available.

Chou *et al.* and later Palix *et al.* used this compiler-based methodology to answer questions about the nature of bugs in the Linux kernel [49,50]. They studied the location of bugs across different subsections of the kernel, bug distribution properties, and bug clustering. At a high-level, this type of analyses and characterization is similar to how the feature vectors are used in Section 4.2 to study the nature of various Android System Services. They also performed differential analysis among different kernel versions to learn how long bugs live in a particular kernel and how different kernels (e.g., OpenBSD vs Linux) compare. Again, at a high level, this is similar to the vendor comparison discussed in Section 4.3. Thus, the feature database described in Section 4.4 could be used to answer similar questions about the Android codebase in future work.

Zombie features are kernel features which cannot be enabled or disabled at all because of inconsistencies between the kernel implementation and its representation in the configuration tool. [51] extracts features from C preprocessor blocks in kernel source and compares them with a corresponding model generated from the kernel configuration tool to identify inconsistencies and find zombie features. [52] maps configuration options to related source code files, enabling identification of sources affected by a change to configuration options. Both of these techniques could be applied to Android's Linux kernel, and the results would complement the various Framework analyses described here.

6.2 Resource Protection

A number of previous efforts have addressed the problem of untrusted apps having access to sensitive or private information. Some of these address specific types of data, such as location, while others look for more general solutions. These works can be classified into broad categories of inspection, permissions, and isolation.

Approaches using inspection, including IPC Inspection [76], Quire [92] and TaintDroid [103], have provided more access control and tracking of call chains, IPC messages, and information flows, respectively. Although these may identify or prevent privilege escalation and confused deputy attacks among apps, they do not address an app's direct access to resources it already has adequate permissions for, as PINPOINT does. While TaintDroid can identify restricted information flows, it has no provisions for preventing them.

AppFence [104] leverages TaintDroid monitoring to enable data substitution and blocking. For information resources, such as location and IMEI, the resulting capability is similar to some of PINPOINT's basic namespaces. However, AppFence cannot control the semantics of functional resources, such as we demonstrated with the input method namespace. Furthermore, AppFence's substitution and blocking capabilities affect information resources for the entire platform rather than being selective for individual apps as is the case in the System Services case study.

Mr. Hide [105] added finer-grained permissions to apps by way of byte code rewriting, while APEX [106] introduced context-sensitive run-time permissions. Compac [107] allows different components within apps, such as in-app ads, to have different sets of permissions. The work of Chapter 5 and other isolation approaches are fundamentally different from these, as the previous work strives to enhance inadequate access control mechanisms, while isolation approaches use virtualization to place untrusted apps in containers where sensitive or vulnerable resources are simply not present, or exist with redefined semantics.

Previous work in the area of isolation includes MOSES [108], a framework designed to isolate applications and data for the purpose of protecting sensitive corporate data. While MOSES represents an effective general solution to securing corporate data leaks on mixed-use personal/business devices, it is not very suitable for protecting users' privacy or securing specific resources because of its security profile-centric architecture that forces explicit switching and carries performance penalties.

Two significant isolation approaches that influenced the PINPOINT effort tremendously are Cells[65] and AirBag [64]. Cells leverages Linux Namespaces to allow multiple Android user spaces to run simultaneously on a single hardware platform. Each user space, or virtual phone (VP), is isolated in a combination of separate Linux Namespaces for file system paths (mount namespace), process identifiers (pid namespace), IPC identifiers (System V IPC namespace), network interface names (network namespace), user names (userid namespace), and hardware devices (a new Linux namespace introduced in the work). Cells introduces the concept of a foreground and multiple background phones that are isolated from each other so that malicious or buggy apps in one VP cannot affect others. Isolation in Cells is thus achieved at the virtual phone boundary.

AirBag also leverages Linux Namespaces, but achieves isolation at the native runtime boundary. This is accomplished by instantiating a separate app runtime that has virtually no interaction with the original native runtime. Each isolated runtime contains its own copies of key service processes and daemons, such as `vold`, `binder` and `servicemanager` that are launched in separate namespaces as compared with the normal runtime. Thus, an untrusted app “sees” an entirely different set of services, binder objects, file paths, etc. through the lens of its decoupled runtime. The untrusted app cannot communicate with apps in different runtimes, and the system resources it can view and control are completely dictated by the isolated runtime. Condroid[109] improves on AirBag’s design by restoring binder communications via virtual binders and increasing efficiency by enabling many System Services to be shared among runtimes instead of duplicated.

While Cells, Airbag and Condroid provide excellent general-purpose isolation, their designs are complex, burden the system, and somewhat intrusive from the user’s point of view. For example, all three require special modifications to numerous shared hardware drivers, duplication of system processes and resources not related to the security goals, and introduce significant usability restrictions. Although they leverage lightweight Linux Namespace isolation, key benefits of Namespaces (Table 5.2) are lost when the Android Framework is added on top since many fundamental aspects of Android’s open design are broken by the kernel-level isolation. Fixing these problems greatly complicates the designs. Thus, PINPOINT’s main difference from these works is the deliberate choice *not* to provide a general-purpose solution, but rather one that addresses specific security goals by directly isolating the specific Framework objects associated with the security goals. For these specific cases, PINPOINT is simpler, less burdensome, and more usable.

Finally, as location data is widely viewed as having serious privacy implications, there are numerous works specific to improving location privacy. LP-Guardian [110], LISA [111], and Koi [112] are examples of these. While each is effective for controlling or preventing the use of location data, they are not generally applicable to other resources as PINPOINT is. As the case study on PINPOINTing System Services demonstrates, if the point of virtualization is chosen wisely, the resulting isolation capability is flexible enough to apply to classes of resources rather than only specific ones as these works do.

Chapter 7

Conclusion

Understanding a system's design tenets and use cases, as well as its security architecture and implementation details, are important prerequisites to proposing useful improvements and making sound modifications. In the case of Android, its unique open architecture, emphasis on usability, and scattered access control points make it very different from previously-seen systems. In spite of its open source roots, gaining an understanding of Android security is difficult because of the sheer complexity of the code base, as well as the fact that most Android devices in use run customized, proprietary versions of the platform.

This dissertation introduces a simplified model of access control that can be used to systematically analyze and understand access controls as they relate to different aspects of the system. The systematic process was used in the first and only evaluation of Android's multiuser framework. This comprehensive evaluation uncovered a diversity of issues with the framework, from simple access control omissions to pervasive problems calling into doubt the suitability of the original architecture for supporting multiple users.

Although a labor-intensive manual investigation such as the one applied to the multiuser framework may be fruitful, it is hardly suited to rapid assessment of the new Android versions that appear every few months, or the plethora of custom, closed-source images that are constantly surfacing. Thus, the next part of the dissertation contributes an access control feature extraction technique which automates the characterization by way of static analysis libraries and database techniques. Unlike other static analysis-based approaches, which typically slice the problem in order to achieve great accuracy within a limited scope, the approach here trades off accuracy in favor of a broad scope that can encompass entire subsystems of many images in a relatively short time. The technique was applied in a case study of System Services from 19 real-world system images. The resulting database of over 35,000 methods represents an interactive resource that can be used to quickly identify and isolate problematic areas of a service or in vendor customizations. The efficacy of the feature vector approach was demonstrated in a case study that identified a number of real vulnerabilities and inconsistencies in actual images. It is hoped that this database can be used by system designers as well as vendors to identify problems before new images are released to users.

Finally, the insights gained enabled the development of a novel means to employ virtualization and isolation to solve the problem of protecting sensitive resources and vulnerable services from untrusted apps, while not diminishing Android's open design and strong emphasis on usability and performance. Known as *PINPOINT*, the hypovisor-based solution isolates only the minimum necessary objects in order to achieve stated security goals. As a new system framework feature, it allows problems not directly fixable by end-users, such as those mentioned above, to be addressed in a flexible, lightweight way, without sacrificing usability or performance. The prototype demonstrated the effectiveness of the solution with a case study on four System Services.

Appendices

Appendix A

Android Security Statements

Table A.1 contains the results of a manual parsing of Android documentation to extract security-related statements. These statements were one of several sources that formed the basis for the access control analysis described in Section 4.2. The following documents were included (index corresponds to entries in “source index” column of table):

- 1.1: Security overview [113]
- 1.2: System and kernel security [114]
- 1.3: Application security [115]
- 1.4: Security updates and resources [116]
- 1.5: Security enhancements in Android 5.0 [117]
- 1.6: Security enhancements in Android 4.4 [118]
- 1.7: Security enhancements in Android 4.3 [119]
- 1.8: Security enhancements in Android 4.2 [120]
- 1.9: Security enhancements in Android 1.5 through 4.1 [121]

Table A.1: Android security statements.

Feature / spec	Source index	Object	Enforcement	Assignment
Core OS applications do not request permissions from user	1.3	services, protected APIs	Permission, UID	build
Mandatory access control environment	1.2	files, processes	UID/PID, security context	creation of object
`export=false` by default	1.8	Application components	permission	installation
Application-defined permissions	1.1	intent filters	permission	Installation
Protect user data	1.1	Application data	UID	Installation
Ensure user A does not exhaust user B's CPU resources	1.2	cgroups	UID	Installation
Linux permissions apply to traditional UNIX-like communication mechanisms	1.3	fd, sockets, signals	UID	Installation
Ensure user A does not exhaust user B's CPU devices, e.g., telephony, GPS, bluetooth	1.2	file lock	UID	Installation
Prevent user A from reading user B's files	1.2	Linux DAC	UID	Installation
Ensure user A does not exhaust user B's memory	1.2	Memory isolation	UID	Installation

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Applications cannot access modem AT commands	1.3	modem	UID	Installation
Provide application isolation	1.1	Process & memory	UID	Installation
Application sandbox	1.1	Process & memory	UID	Installation
Process isolation	1.2	Process & memory	UID	Installation
Isolate user resources from one another	1.2	Process & memory	UID	Installation
Intentional lack of APIs to restrict capabilities and access to sensitive functionality (e.g., SIM)	1.3	services	UID	Installation
Low-level SIM card access is not available to third-party apps	1.3	sim	UID	Installation
User-based permissions model	1.2	uids	UID	Installation
Sensitive resources protected by separation of roles (e.g., storage)	1.3	storage	UID, namespaces, bind mount	Installation
Kernel-level application sandbox	1.2	Process & memory	UID/PID	Installation
Sensitive APIs protected with permissions	1.3	API functionality	permission	manifest permission

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Cost-sensitive APIs are protected APIs	1.3	API functionality	permission	manifest permission
Personal information APIs are protected APIs	1.3	API functionality	permission	manifest permission
Application developer specifies which permissions are needed	1.3	Application components	permission	manifest permission
Applications may declare/define their own permissions for other applications to use	1.3	Application components	permission	manifest permission
Camera is a protected API	1.3	camera	permission	manifest permission
System content providers with user data have been created with clearly identified permissions	1.3	data	permission	manifest permission
In-App Billing is a cost-sensitive API	1.3	financial	permission	manifest permission

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Location data (GPS) is a protected API	1.3	location	permission	manifest permission
Restricts access to device metadata	1.3	metadata	permission	manifest permission
Network/data is a cost-sensitive API	1.3	network	permission	manifest permission
NFC access is a cost-sensitive API	1.3	nfc	permission	manifest permission
Telephony functions is a protected API	1.3	phone	permission	manifest permission
Telephony is a cost-sensitive API	1.3	phone	permission	manifest permission
By default, application can only access a limited range of system services	1.3	services	permission	manifest permission

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
SMS/MMS functions is a protected API	1.3	sms	permission	manifest permission
SMS/MMS is a cost-sensitive API	1.3	sms	permission	manifest permission
Bluetooth functions is a protected API	1.3	bluetooth h/w	UID	manifest permission
Network/data connections is a protected API	1.3	network	UID	manifest permission
Applications that accumulate user data can use permission checks to protect this data from other applications	1.3	Application components, application data, external storage	UID, permission	manifest permission
User must grant explicit permission to third-party applications requesting use of sensitive data input devices (e.g., camera, mic, GPS)	1.3	input devices	Permission, UID	manifest permission, installation
Secure IPC	1.1	IPC message	capability	start of IPC

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
ADB authentication	1.7	adb, shell	keypair	User-based
Users can turn off some functionality globally (e.g., GPS, radio, WiFi)	1.3	hardware device	on/off	User-based
User-granted permissions	1.1	manifest permissions	permission	User-based
User must grant explicit permission to third-party applications requesting use of cost-sensitive APIs	1.3	API functionality	user	User-based
Prior to installation, users are shown message about permissions application is requesting	1.3	services, protected APIs, system resources	user	User-based
User notification and block/allow choice is provided if application attempts to send SMS to short code that uses premium services	1.3	sms	user	User-based
Device access can require user password	1.2	Device UI	user	User-based
Application verification (Verify Apps)	1.8		installation	User-based
Capability bounding âĂŞ PR_CAPBSET_DROP	1.7		zygote	
No new privs âĂŞ PR_SET_NO_NEW_PRIVS	1.7		zygote	

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Certificate pinning	1.8		domain	certificate
Protect system resources	1.1			
Application signing	1.1			
Remove insecure parts of kernel	1.2			
Identifies and isolates application resources	1.2			
By default, application cannot interact with one another	1.2			
Applications have limited access to the operating system	1.2			
Operating system libraries run within application sandbox	1.2			
Application framework components run within application sandbox	1.2			
Application runtime runs within application sandbox	1.2			
All applications run within application sandbox	1.2			
No restriction on applications to enforce security	1.2			
Native code is just as secure as interpreted code	1.2			

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Memory corruption errors allow arbitrary code execution only in the context of the corrupted application	1.2			
Only way to break out of the application sandbox is to compromise the Linux kernel	1.2			
System partition is read-only	1.2			
Safe mode boot with no third-party applications	1.2			
By default, files created by one app cannot be read or altered by another app	1.2			
Provides cryptographic APIs for use by applications	1.2			
Provides access to system credential storage for private keys and cert chains	1.2			
By default, only kernel and small subset of core applications run with root permissions	1.2			
Does not prevent an application with root permissions from modifying OS, kernel or any other application	1.2			

Android security statements (continued).

Feature / spec	**Source index**	**Object**	**Enforcement**	**Assignment**
Root has full access to all applications and all application data	1.2			
Users have the ability to unlock bootloader	1.2			
Can install new OS with physical control and USB	1.2			
Bootloader erases user data as part of unlock procedure	1.2			
Full filesystem encryption uses the device password to protect encryption key	1.2			
Provides full filesystem encryption using dmcrypt's AES128 with CBC and ESSIV:SHA256	1.2			
Protects against systematic password guessing of device password	1.2			
Protects against dictionary attacks on device password	1.2			
Device administrator can set password and password complexity rules	1.2			
Provides device administration features	1.2			

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Protected resources are only available through the OS	1.3			
Application defined the capabilities it needs	1.3			
User grants or denies all of an application's requested permissions	1.3			
Permissions apply to application as long as it is installed	1.3			
OEM applications do not request permissions from user	1.3			
Once installed, user is not notified of permissions granted to application	1.3			
Users may view permissions for installed applications	1.3			
Security exceptions are thrown back to applications that attempt to use protected features not declared by the application	1.3			
Permission checks are performed at the lowest possible level	1.3			
Permission attributes define how user is to be informed about permission	1.3			

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Permission attributes defined who is allowed to hold permission	1.3			
Some capabilities are not available to third-party applications	1.3			
Some capabilities may be used by OEM applications	1.3			
Processes can communicate using any traditional UNIX-like mechanisms	1.3			
By default, applications do not have access to logs, history, phone number, hardware/network identification info.	1.3			
Every application must be signed by developer	1.3			
Unsigned applications with be rejected by the package installer	1.3			
Application certificate defines which uid is associated with application	1.3			

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
Applications with matching certificates may share uids if requested by developer	1.3			
Applications may be self-signed	1.3			
Applications may declare permissions that are only available to other applications signed with the same key	1.3			
User may choose to enable application verification	1.3			
Provides DRM	1.3			
Full disc encryption is enabled by default	1.5			
Device password protected against brute-force attacks	1.5			
Device password protected against off-device attacks	1.5			
Sandbox reinforced with SELinux	1.5			
Smartlock trustlets	1.5			
Multi-user, restricted profile, guest mode	1.5			
Update Webview via OTA	1.5			
Cryptography	1.5			

Android security statements (continued).

Feature / spec	**Source index**	**Object**	**Enforcement**	**Assignment**
PIE support (improves ASLR)	1.5			
`FORTIFY_SOURCE`	1.5			
Per-user VPN	1.6			
ECDSA Support	1.6			
Certificate added warnings	1.6			
Certificate pinning	1.6			
No setuid/setgid binaries	1.7			
nosuid in /system	1.7			
Read-only relocation protection (relro)	1.7			
Improved EntropyMixer	1.7			
Premium SMS	1.8			
Always-on VPN	1.8			
installd hardening	1.8			
init scripts `O_NOFOLLOW`	1.8			
`FORTIFY_SOURCE`	1.8			

Android security statements (continued).

Feature / spec	Source index	Object	Enforcement	Assignment
`-fstack-protector`	1.9			
`safe_iop`	1.9			
dlmalloc extensions	1.9			
calloc	1.9			
`-Wformat-security`	1.9			
Hardware NX	1.9			
`mmap_min_addr`	1.9			
ASLR	1.9			
PIE support (improves ASLR)	1.9			
Read-only relocation protection (relro)	1.9			
`dmesg_restrict` enabled	1.9			
`kptr_restrict` enabled	1.9			
Permissions are removed when application is uninstalled	1.3			

Appendix B

Android Image Extraction Procedures

This appendix describes the procedures used in this research for extracting JARs and other system files from Android image files. In general, it was found that every image is different, especially among different commercial vendors that may use different formats or protection mechanisms. Also, the tools used are u Thus, development of a fully-automated process is unlikely, unless the goal is to process a large number of very similar images. However, it was also apparent that many of the differences disappear once the raw image is unpacked and mounted. The final processing then becomes a matter of using the right tool depending on what type of files are present (OAT, ODEX, DEX, JAR, etc.), which is almost entirely dependent on the version of Android (i.e., KitKat, Lollipop, etc.).

B.1 AOSP, Android KitKat version 4.4.4

Extract the contents of the downloaded tar file (i.e., `tar xz`). Convert `system.img` using `simg2img`, mount converted image to system folder, then `system/framework` contains the framework JARs and ODEX files. Use `baksmali`, `smali`, and `dex2jar` to convert them to JARs.

```
simg2img system.img u-system.img
mkdir system
```

```
sudo mount -o loop system.img system/
cd system
baksmali -d framework/ -x framework/services.jar
smali ./out
dex2jar out.dex
mv out-dex2jar.jar new-services.jar
```

B.2 AOSP, Android Lollipop version 5.x

Extract the contents of the downloaded tar file (i.e., `tar xz`). Convert `system.img` using `simg2img`, mount converted image to system folder, then `system/framework/arm` contains `boot.oat` and framework ODEX files. Use `dextra` to extract DEX files from `boot.oat` and and framework ODEX files, and `dex2jar` to convert framework ODEX files to JARs. NOTE: `baksmali` probably will not work on ODEX files prior to version 56, which roughly corresponds to Marshmallow [122]. Use `dextra` as below or alternate method in Windows.

```
simg2img system.img u-system.img
mkdir system
sudo mount -o loop system.img system/
cd system
dextra -dextract framework/arm/boot.oat
dextra -dextract framework/arm/services.odex
dex2jar system@framework@framework.jar@classes.dex
dex2jar system@framework@services.jar@classes.dex
mv system@framework@framework.jar@classes-dex2jar.jar framework.jar
mv system@framework@services.jar@classes-dex2jar.jar services.jar
```

Alternate method of deodex: Use *JoelDroid Lollipop Batch Deodexer* [123] on system folder.

B.3 AOSP, Android Marshmallow version 6.x

Extract the contents of the downloaded tar file (i.e., `tar xz`). Convert `system.img` using `simg2img`, mount converted image to system folder, then `system/framework/arm` contains `boot.oat` and `system/framework/oat/arm` contains framework ODEX files. Use `dextra` to extract DEX files from `boot.oat` and framework ODEX files, and `dex2jar` to convert DEX files to JARs.

```
simg2img system.img u-system.img
mkdir system
sudo mount -o loop system.img system/
cd system
dextra -dextract framework/arm/boot.oat
dextra -dextract framework/oat/arm/services.odex
dex2jar system@framework@framework.jar@classes.dex
dex2jar system@framework@services.jar@classes.dex
mv system@framework@framework.jar@classes-dex2jar.jar framework.jar
mv system@framework@services.jar@classes-dex2jar.jar services.jar
```

B.4 CyanogenMod 11-20150901, Android KitKat version 4.4.4

Unzip the downloaded file. `system/framework` contains the framework JARs in DEX format. Use `baksmali`, `smali`, and `dex2jar` to convert them to standard JARs.

```
baksmali -d framework/ -x framework/services.jar
smali ./out
dex2jar out.dex
mv out-dex2jar.jar services.jar
```

B.5 CyanogenMod 12.1-20151121, Android Lollipop version 5.1.1

Unzip the downloaded file. Convert `system.new.dat` to `system.img` using `sdat2img`. Mount image to system folder, then `system/framework` contains the framework JARs. Use `baksmali`, `smali`, and `dex2jar` to convert them to JARs.

```
sdat2img system.transfer.list system.new.dat system.img
mkdir m-system
sudo mount -o loop system.img m-system/
cd m-system
baksmali -d framework/ -x framework/services.jar
smali ./out
dex2jar out.dex
mv out-dex2jar.jar new-services.jar
```

B.6 Xiaomi MIUI, Android KitKat version 4.4.4

Unzip the download, `system/framework` contains the framework JARs. Use `baksmali`, `smali`, `dex2jar` to convert them to JARs.

```
baksmali -d framework/ -x framework/services.jar
smali ./out
dex2jar out.dex
mv out-dex2jar.jar services.jar
```

B.7 FireOS 32.4.6.5, Android KitKat version 4.4.4

Extracting the JARs from this image is problematic and not completely working. This version of FireOS contains ODEX files coded as version 39, which is different than other KitKat images and not supported by the Smali tools. Hexedit was used to rewrite the version field to 36, which enabled **baksmali** to run, but it produced many errors. Inspection of the resulting JARs revealed that they are incomplete and have many classed replaced with **//INTERNAL ERROR**. Otherwise, the process is the same as other KitKat images.

Rename the **.bin** file to **.zip** and unzip. **system/framework** contains the framework JARs in DEX format. Use a hexeditor to change the version fields to 36. Use **baksmali**, **smali**, and **dex2jar** to convert them to standard JARs.

B.8 FireOS 37.5.2.2, Android Lollipop version 5.0.2

Rename the **.bin** file to **.zip** and unzip. Convert **system.new.dat** to **system.img** using **sdat2img**. Mount image to system folder, then **system/framework/arm** contains **boot.oat** and framework ODEX files. Use **dextra** to extract DEX files from **boot.oat** and and framework ODEX files. **dex2jar** to convert framework ODEX files to JARs. NOTE: **baksmali** probably will not work on ODEX files prior to version 56, which roughly corresponds to Android Marshmallow version 6.x [122]. Use **dextra** as shown below or alternate method in Windows.

```
sdat2img system.transfer.list system.new.dat system.img
mkdir m-system
sudo mount -o loop system.img m-system/
cd m-system
dextra -dextract framework/arm/boot.oat
dextra -dextract framework/arm/services.odex
```

```
dex2jar system@framework@framework.jar@classes.dex
dex2jar system@framework@services.jar@classes.dex
mv system@framework@framework.jar@classes-dex2jar.jar framework.jar
mv system@framework@services.jar@classes-dex2jar.jar services.jar
```

Alternate method of deodex: Use *JoelDroid Lollipop Batch Deodexer* [123] on system folder.

B.9 LG, Android KitKat version 4.4.2

Use *LG Firmware Extract* [**lg_extract**] on downloaded file to extract KDZ, DZ, and merge `system-bin` into `system.img`. Mount `system.img` to system folder, then `system/framework` contains the framework JARs and ODEX files. Use `baksmali`, `smali`, `dex2jar` to convert them to JARs.

```
mkdir system
sudo mount -o loop system.img system/
cd system
baksmali -d framework/ -x framework/services.jar
smali ./out
dex2jar out.dex
mv out-dex2jar.jar new-services.jar
```

B.10 LG, Android Lollipop version 5.x

Use *LG Firmware Extract* [**lg_extract**] on downloaded file to extract KDZ, DZ, and merge `system-bin` into `system.img`. Mount `system.img` to system folder, then `system/framework/arm` contains `boot.oat` and framework ODEX files. Use `dextra` to extract DEX files from `boot.oat` and and framework ODEX files, and `dex2jar` to convert framework ODEX files to JARs. NOTE: **baksmali** probably will not work on ODEX files prior to version 56, which roughly corresponds to Android Marshmallow version 6.x [122]. Use **dextra** as below or alternate method in Windows.

```
mkdir system
sudo mount -o loop system.img system/
cd system
dextra -dextract framework/arm/boot.oat
dextra -dextract framework/arm/services.odex
dex2jar system@framework@framework.jar@classes.dex
dex2jar system@framework@services.jar@classes.dex
mv system@framework@framework.jar@classes-dex2jar.jar framework.jar
mv system@framework@services.jar@classes-dex2jar.jar services.jar
```

Alternate method of deodex: Use *JoelDroid Lollipop Batch Deodexer* [123] on system folder.

B.11 HTC RUU, Android Jellybean version 4.2.x

Get **rom.zip** from RUU exe file by either 1) running RUU in Windows, finding temporary location of extracted **rom.zip** and copy it; or 2) use **unruu** [124] in Linux to extract **rom.zip**. Use **ruuveal** [125] to decrypt **rom.zip**, then use standard methods to extract **system.img**, convert (if necessary), mount. Finally, use the standard **baksmali-smali-dex2jar** toolchain to produce JARs.

B.12 HTC RUU, Android KitKat and later versions

Unfortunately, KitKat and later ROMs from HTC use strong encryption and known methods, such as **ruuveal**, do not work for decrypting the ROM archive.

Appendix C

Permission Configuration Extraction

This appendix contains source code of the scripts written to capture the permissions and associated protection levels from unpacked Android images. **xtract_perms.py**, shown in Listing C.1, is run from the root folder of an unpacked image. It uses **apktool** to decode each APK found in the image and store its manifest. Once all manifest files are found, **xtract_perms_config.py** is called to search each manifest for permission names and their protection level. This combined information is written to a text file which is read by *FeatureExtraction*.

*Listing C.1: **xtract_perms.py***

```
#!/usr/bin/python

import os
import subprocess

decode_dir = "./apk_decode"
exclude = set(["WORKING"])
top = "../"

try:
    os.makedirs(decode_dir)
except OSError:
    if not os.path.isdir(decode_dir):
        raise

for subdir, dirs, files in os.walk(top, topdown=True):
    dirs[:] = [d for d in dirs if d not in exclude]
    for file in files:
        #print os.path.join(subdir, file)
        filepath = subdir + os.sep + file

        if filepath.endswith(".apk"):
            print ("Decoding: " + filepath)
            out_dir = decode_dir + "/" + file
            subprocess.call(["apktool", "d", "-s", "-o", out_dir, filepath])
```

```

print "apktool finished; extracting permissions from manifest files..."

fd = open("allpermsV2.txt", "w")
subprocess.call(["xtract_perm_config.py", decode_dir], stdout=fd, shell=False)
```

Listing C.2: `xtract_perm_config.py`

```
#!/usr/bin/python

import sys
from xml.dom.minidom import parse
import xml.dom.minidom
import os
from os import path
import csv

# argv[1] is name of directory containing the manifest files
inputDir = sys.argv[1]

images = []
class Image():
        name = ""
        files = ""
        def __init__(self, name):
                self.files = []
                self.name = name

for path, subdirs, filenames in os.walk(inputDir):
        if not (path == inputDir):
                image = Image(path)
        for filename in [f for f in filenames if (f.endswith(".xml")) ]:
                image.files.append(os.path.join(path, filename))
        if not (path == inputDir):
                images.append(image)

for image in images:
        for filename in image.files:

                try:
                        DOMTree = xml.dom.minidom.parse(filename)
                except:
                        continue

                manifest = DOMTree.documentElement

                permissions_m = manifest.getElementsByTagName("permission")

                for permission_m in permissions_m:
                        name = ""
                        level ="-1" #"No Level Specified"
                        if permission_m.hasAttribute("android:name"):
                                name = permission_m.getAttribute("android:name")

                        if permission_m.hasAttribute("android:protectionLevel"):
                                level = permission_m.getAttribute("android:protectionLevel")

                                        if (level == "signature") or ("Signature" in level) or ("system" in level) or ("System" in level):
                                        level = "3"
                                if level == "dangerous":
                                        level = "2"
                                if level == "normal":
                                        level = "1"
```

```
                            #print " Permission Name : " + name + " Protection Level : " + level
                            sys.stdout.write(name)
                            sys.stdout.write(’:’)
                            sys.stdout.write(level)
                            sys.stdout.write(’;’)
```

Appendix D

Power Query Import Script

This appendix contains the PQFL script used to import a directory tree of feature vector CSVs into Microsoft®Excel®. The root of the CSV filename is used to identify the System Service corresponding to the feature vector content.

Listing D.1: PQFL script to import multiple feature vector CSV files into Microsoft® Excel® for analysis.

```
let
    Source = Folder.Files("E:\CSVs"),
    #"Removed Other Columns" = Table.SelectColumns(Source,{"Content", "Name", "Folder Path"
        }),
    #"Reordered Columns" = Table.ReorderColumns(#"Removed Other Columns",{"Content", "Folder
         Path", "Name"}),
    #"Filtered Rows1" = Table.SelectRows(#"Reordered Columns", each true),
    #"Split Column by Delimiter" = Table.SplitColumn(#"Filtered Rows1","Folder Path",
        Splitter.SplitTextByDelimiter("\", QuoteStyle.Csv),{"Folder Path.1", "Folder Path.2"
        , "Folder Path.3", "Folder Path.4"}),
    #"Removed Columns" = Table.RemoveColumns(#"Split Column by Delimiter",{"Folder Path.1",
        "Folder Path.2", "Folder Path.4"}),
    #"Split Column by Delimiter1" = Table.SplitColumn(#"Removed Columns","Name",Splitter.
        SplitTextByDelimiter(".", QuoteStyle.Csv),{"Name.1", "Name.2"}),
    #"Removed Columns1" = Table.RemoveColumns(#"Split Column by Delimiter1",{"Name.2"}),
    #"Renamed Columns" = Table.RenameColumns(#"Removed Columns1",{{"Folder Path.3", "IMAGE"
        }, {"Name.1", "SERVICE"}}),
    #"Add and Import" = Table.AddColumn(Source, "Custom", each Table.PromoteHeaders(Csv.
        Document([Content],[Delimiter=",", Encoding=1252]))),
    #"Removed Columns2" = Table.RemoveColumns(#"Add and Import",{"Extension", "Date accessed
        ", "Date modified", "Date created"}),
    #"Expanded Custom" = Table.ExpandTableColumn(#"Removed Columns2", "Custom", {"METHOD", "
        is_AIDL", "getCallingUid(", "getCallingPid(", "clearCallingIdentity(", "
        restoreCallingIdentity(", "checkPermission(", "checkCallingOrSelfPermission(", "
        checkCallingPermission(", "enforcePermission(", "enforceCallingPermission(", "
        enforceCallingOrSelfPermission(", "security_exception_raised", "permission_normal",
        "permission_dangerous", "permission_sig", "permission_undef"}, {"Custom.METHOD", "
        Custom.is_AIDL", "Custom.getCallingUid(", "Custom.getCallingPid(", "Custom.
        clearCallingIdentity(", "Custom.restoreCallingIdentity(", "Custom.checkPermission(",
         "Custom.checkCallingOrSelfPermission(", "Custom.checkCallingPermission(", "Custom.
        enforcePermission(", "Custom.enforceCallingPermission(", "Custom.
        enforceCallingOrSelfPermission(", "Custom.security_exception_raised", "Custom.
        permission_normal", "Custom.permission_dangerous", "Custom.permission_sig", "Custom.
        permission_undef"}),
    #"Removed Columns3" = Table.RemoveColumns(#"Expanded Custom",{"Attributes"}),
    #"Split Column by Delimiter2" = Table.SplitColumn(#"Removed Columns3","Folder Path",
        Splitter.SplitTextByDelimiter("\", QuoteStyle.Csv),{"Folder Path.1", "Folder Path.2"
        , "Folder Path.3", "Folder Path.4"}),
    #"Changed Type" = Table.TransformColumnTypes(#"Split Column by Delimiter2",{{"Folder
        Path.1", type text}, {"Folder Path.2", type text}, {"Folder Path.3", type text}, {"
        Folder Path.4", type text}}),
    #"Removed Columns4" = Table.RemoveColumns(#"Changed Type",{"Folder Path.1", "Folder Path
        .2"}),
    #"Renamed Columns1" = Table.RenameColumns(#"Removed Columns4",{{"Folder Path.3", "IMAGE"
        }}),
    #"Removed Columns5" = Table.RemoveColumns(#"Renamed Columns1",{"Folder Path.4"}),
    #"Filtered Rows" = Table.SelectRows(#"Removed Columns5", each [Custom.is_AIDL] = "1"),
    #"Split Column by Delimiter3" = Table.SplitColumn(#"Filtered Rows","Custom.METHOD",
        Splitter.SplitTextByDelimiter(",", QuoteStyle.Csv),{"Custom.METHOD.1", "Custom.
        METHOD.2", "Custom.METHOD.3"}),
    #"Changed Type1" = Table.TransformColumnTypes(#"Split Column by Delimiter3",{{"Custom.
        METHOD.1", type text}, {"Custom.METHOD.2", type text}, {"Custom.METHOD.3", type text
        }}),
    #"Renamed Columns2" = Table.RenameColumns(#"Changed Type1",{{"Custom.METHOD.3", "METHOD"
        }}),
    #"Split Column by Delimiter4" = Table.SplitColumn(#"Renamed Columns2","Name",Splitter.
        SplitTextByDelimiter(".", QuoteStyle.Csv),{"Name.1", "Name.2"}),
    #"Changed Type2" = Table.TransformColumnTypes(#"Split Column by Delimiter4",{{"Name.1",
        type text}, {"Name.2", type text}}),
    #"Removed Columns6" = Table.RemoveColumns(#"Changed Type2",{"Name.2"}),
    #"Reordered Columns1" = Table.ReorderColumns(#"Removed Columns6",{"Content", "IMAGE", "
        Name.1", "Custom.METHOD.1", "Custom.METHOD.2", "METHOD", "Custom.is_AIDL", "Custom.
        getCallingUid(", "Custom.getCallingPid(", "Custom.clearCallingIdentity(", "Custom.
        restoreCallingIdentity(", "Custom.checkPermission(", "Custom.
        checkCallingOrSelfPermission(", "Custom.checkCallingPermission(", "Custom.
        enforcePermission(", "Custom.enforceCallingPermission(", "Custom.
        enforceCallingOrSelfPermission(", "Custom.security_exception_raised", "Custom.
        permission_normal", "Custom.permission_dangerous", "Custom.permission_sig", "Custom.
        permission_undef"}),
    #"Split Column by Delimiter5" = Table.SplitColumn(#"Reordered Columns1","Custom.METHOD.1
        ",Splitter.SplitTextByDelimiter(" ", QuoteStyle.Csv),{"Custom.METHOD.1.1", "Custom.
        METHOD.1.2", "Custom.METHOD.1.3", "Custom.METHOD.1.4"}),
    #"Changed Type3" = Table.TransformColumnTypes(#"Split Column by Delimiter5",{{"Custom.
        METHOD.1.1", type text}, {"Custom.METHOD.1.2", type text}, {"Custom.METHOD.1.3",
        type text}, {"Custom.METHOD.1.4", type text}}),
    #"Removed Columns7" = Table.RemoveColumns(#"Changed Type3",{"Custom.METHOD.1.1"}),
    #"Renamed Columns3" = Table.RenameColumns(#"Removed Columns7",{{"Custom.METHOD.1.2", "
        METHOD TYPE"}}),
    #"Removed Columns8" = Table.RemoveColumns(#"Renamed Columns3",{"Custom.METHOD.1.3", "
        Custom.METHOD.1.4"}),
    #"Renamed Columns4" = Table.RenameColumns(#"Removed Columns8",{{"Custom.METHOD.2", "
        CLASSPATH"}, {"Custom.is_AIDL", "isAIDL"}, {"Custom.getCallingUid(", "getCallingUid"
        }, {"Custom.getCallingPid(", "getCallingPid"}, {"Custom.clearCallingIdentity(", "
        clearCallingIdentity"}, {"Custom.restoreCallingIdentity(", "restoreCallingIdentity"
        }, {"Custom.checkPermission(", "checkPermission"}, {"Custom.
        checkCallingOrSelfPermission(", "checkCallingOrSelfPermission"}, {"Custom.
        checkCallingPermission(", "checkCallingPermission"}, {"Custom.enforcePermission(", "
        enforcePermission"}, {"Custom.enforceCallingPermission(", "enforceCallingPermission"
        }, {"Custom.enforceCallingOrSelfPermission(", "enforceCallingOrSelfPermission"}, {"
        Custom.security_exception_raised", "securityException"}, {"Custom.permission_normal"
        , "permissionNormal"}, {"Custom.permission_dangerous", "permissionDangerous"}, {"
        Custom.permission_sig", "permissionSig"}, {"Custom.permission_undef", "
        permissionUndef"}, {"Name.1", "SERVICE"}}),
    #"Changed Type4" = Table.TransformColumnTypes(#"Renamed Columns4",{{"isAIDL", Int64.Type
        }, {"getCallingUid", Int64.Type}, {"getCallingPid", Int64.Type}, {"
        clearCallingIdentity", Int64.Type}, {"restoreCallingIdentity", Int64.Type}, {"
        checkPermission", Int64.Type}, {"checkCallingOrSelfPermission", Int64.Type}, {"
        checkCallingPermission", Int64.Type}, {"enforcePermission", Int64.Type}, {"
        enforceCallingPermission", Int64.Type}, {"enforceCallingOrSelfPermission", Int64.
        Type}, {"securityException", Int64.Type}, {"permissionNormal", Int64.Type}, {"
        permissionDangerous", Int64.Type}, {"permissionSig", Int64.Type}, {"permissionUndef"
        , Int64.Type}})
in
    #"Changed Type4"
```

Appendix E

AOSP System Service Pivot Analysis

This appendix contains the results of a pivot analysis of System Service feature vectors from various versions of official “stock” AOSP images. Blank columns indicate either that the service is not present in that particular version of AOSP, or that *FeatureExtraction* was unable to process the JAR/class for that image.

The *Count of METHOD* rows show the total number of AIDL methods in the service. *Sum* rows show the total number of occurances of each feature among the AIDL methods. *Sum of None* reveals the total number of AIDL methods that have no access control features.

Table E.1: Feature counts for each System Service in AOSP baseline images.

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
AccessibilityManagerService							
Count of METHOD	10	10	11	11	11	11	11
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	1	1	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	9	9	10	10	10	10	10
AccountManagerService							
Count of METHOD	24	30	35	35	35	37	39
Sum of getCallingUid	21	25	29	29	29	31	32
Sum of getCallingPid	19	19	23	23	23	24	23
Sum of clearCallingIdentity	21	21	25	25	25	27	27
Sum of restoreCallingIdentity	21	21	25	25	25	27	27
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	2	5	5	5	6	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	2	5	6	6	6	6	7
ActivityManagerService							
Count of METHOD	46	47	16	16	39	19	25
Sum of getCallingUid	26	25	7	7	23	9	9
Sum of getCallingPid	26	24	7	7	22	9	9
Sum of clearCallingIdentity	15	11	4	4	10	5	5
Sum of restoreCallingIdentity	15	11	4	4	10	5	5
Sum of permissionNormal	0	0	0	0	0	0	1
Sum of permissionDangerous	1	1	0	0	3	0	0
Sum of permissionSig	26	24	7	7	22	9	9
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	18	20	9	9	13	10	15

continued...

Feature counts for each System Service in AOSP baseline images (continued).

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
ActivityManagerService$CpuBinder							
Count of METHOD	28	31	32	32	32	32	33
Sum of getCallingUid	2	2	2	2	2	2	2
Sum of getCallingPid	1	1	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	26	28	29	29	29	29	30
ActivityManagerService$DbBinder							
Count of METHOD	28	31	32	32	32	32	33
Sum of getCallingUid	2	2	2	2	2	2	2
Sum of getCallingPid	1	1	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	26	28	29	29	29	29	30
ActivityManagerService$GraphicsBinder							
Count of METHOD	28	31	32	32	32	32	33
Sum of getCallingUid	2	2	2	2	2	2	2
Sum of getCallingPid	1	1	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	26	28	29	29	29	29	30
ActivityManagerService$MemBinder							

continued…

Feature counts for each System Service in AOSP baseline images (continued).

Service	Image AOSP-4.2.2	AOSP-4.4.4	AOSP-5.0	AOSP-5.0.1	AOSP-5.0.2	AOSP-5.1.1	AOSP-6.0
Count of METHOD	28	31	32	32	32	32	33
Sum of getCallingUid	2	2	2	2	2	2	2
Sum of getCallingPid	1	1	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	26	28	29	29	29	29	30
ActivityManagerService$PermissionController							
Count of METHOD	1	1	1	1	1	1	3
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	2
ActivityManagerService$ProcessInfoService							
Count of METHOD	0	0	0	0	0	0	1
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	1
AlarmManagerService							
Count of METHOD	7	6	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of getCallingUid	0	1	0	0	0	0	0
Sum of getCallingPid	0	1	0	0	0	0	0
Sum of clearCallingIdentity	1	1	0	0	0	0	0
Sum of restoreCallingIdentity	1	1	0	0	0	0	0
Sum of permissionNormal	1	1	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	2	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	5	2	0	0	0	0	0
AlarmManagerService$2							
Count of METHOD	0	0	5	5	5	5	6
Sum of getCallingUid	0	0	1	1	1	1	2
Sum of getCallingPid	0	0	1	1	1	1	2
Sum of clearCallingIdentity	0	0	1	1	1	1	1
Sum of restoreCallingIdentity	0	0	1	1	1	1	1
Sum of permissionNormal	0	0	1	1	1	1	1
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	2	2	2	2	2
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	2
AppOpsService							
Count of METHOD	0	12	16	16	16	16	19
Sum of getCallingUid	0	3	6	6	6	6	8
Sum of getCallingPid	0	3	4	4	4	4	5
Sum of clearCallingIdentity	0	0	0	0	0	0	2
Sum of restoreCallingIdentity	0	0	0	0	0	0	2
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	3	4	4	4	4	5
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	9	10	10	10	10	11
AppWidgetService							
Count of METHOD	25	23	0	0	0	0	0
Sum of getCallingUid	4	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of getCallingPid	4	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	21	23	0	0	0	0	0
AppWidgetServiceImpl							
Count of METHOD	0	0	24	24	24	24	25
Sum of getCallingUid	0	0	19	19	19	19	20
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	2	2	2	2	2
Sum of restoreCallingIdentity	0	0	2	2	2	2	2
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	2	2	2	2	2
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	3	3	3	3	3
AssetAtlasService							
Count of METHOD	0	3	3	3	3	3	3
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	3	3	3	3	3	3
AudioService							
Count of METHOD	67	74	69	69	69	78	72
Sum of getCallingUid	7	8	12	12	12	14	18
Sum of getCallingPid	5	5	7	7	7	8	8

continued…

Feature counts for each System Service in AOSP baseline images (continued).

Service	AOSP-4.2.2	AOSP-4.4.4	AOSP-5.0	Image AOSP-5.0.1	AOSP-5.0.2	AOSP-5.1.1	AOSP-6.0
Sum of clearCallingIdentity	1	2	2	2	2	2	2
Sum of restoreCallingIdentity	1	2	2	2	2	2	2
Sum of permissionNormal	5	5	4	4	4	4	3
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	3	1	8	8	8	11	13
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	56	65	50	50	50	55	50
BackupManagerService							
Count of METHOD	22	22	25	25	25	0	0
Sum of getCallingUid	6	6	5	5	6	0	0
Sum of getCallingPid	1	1	0	0	1	0	0
Sum of clearCallingIdentity	4	5	6	6	7	0	0
Sum of restoreCallingIdentity	4	5	6	6	7	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	17	17	19	19	20	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	1	1	2	2	1	0	0
BatteryService							
Count of METHOD	52	54	0	0	0	0	0
Sum of getCallingUid	2	2	0	0	0	0	0
Sum of getCallingPid	1	1	0	0	0	0	0
Sum of clearCallingIdentity	0	1	0	0	0	0	0
Sum of restoreCallingIdentity	0	1	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	50	51	0	0	0	0	0
BatteryService$BinderService							
Count of METHOD	0	0	33	33	33	33	34
Sum of getCallingUid	0	0	2	2	2	2	2
Sum of getCallingPid	0	0	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0

continued...

Feature counts for each System Service in AOSP baseline images (continued).

			Image				
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	30	30	30	30	31
BatteryStatsService							
Count of METHOD	42	47	68	68	68	69	74
Sum of getCallingUid	39	44	62	62	62	63	67
Sum of getCallingPid	39	44	62	62	62	63	67
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	3	0	0	0	0	0	0
Sum of permissionSig	39	47	66	66	66	67	71
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	2	2	2	2	3
BluetoothManagerService							
Count of METHOD	10	11	11	11	11	13	16
Sum of getCallingUid	6	6	6	6	6	6	6
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	5	5	5	5	5	5	5
Sum of restoreCallingIdentity	5	5	5	5	5	5	5
Sum of permissionNormal	0	0	0	0	0	0	7
Sum of permissionDangerous	8	8	8	8	8	8	0
Sum of permissionSig	0	0	0	0	0	0	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	1	2	2	2	2	4	7
ClipboardService							
Count of METHOD	0	0	7	7	7	7	7
Sum of getCallingUid	0	0	6	6	6	6	6
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0

continued...

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	1
CommonTimeManagementService							
Count of METHOD	39	42	43	43	43	43	44
Sum of getCallingUid	2	2	2	2	2	2	2
Sum of getCallingPid	1	1	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	37	39	40	40	40	40	41
ConnectivityService							
Count of METHOD	48	61	66	66	66	65	64
Sum of getCallingUid	8	17	20	20	20	23	24
Sum of getCallingPid	3	3	0	0	0	0	0
Sum of clearCallingIdentity	4	9	6	6	6	6	6
Sum of restoreCallingIdentity	4	9	6	6	6	6	6
Sum of permissionNormal	28	30	26	26	26	24	27
Sum of permissionDangerous	0	0	1	1	1	1	0
Sum of permissionSig	7	18	20	20	20	19	21
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	12	5	9	9	9	7	5
ConsumerIrService							
Count of METHOD	0	3	3	3	3	3	3
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	2	2	2	2	2	2

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	1	1	1	1	1	1
ContentService							
Count of METHOD	23	24	36	36	36	36	37
Sum of getCallingUid	1	4	7	7	7	7	7
Sum of getCallingPid	1	1	1	1	1	1	2
Sum of clearCallingIdentity	19	20	21	21	21	21	22
Sum of restoreCallingIdentity	19	20	21	21	21	21	22
Sum of permissionNormal	13	14	6	6	6	6	6
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	2	2	13	13	13	13	14
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	3	3	14	14	14	14	14
CountryDetectorService							
Count of METHOD	3	3	3	3	3	3	3
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	3	3	3	3	3	3	3
DeviceIdleController$BinderService							
Count of METHOD	0	0	0	0	0	0	15
Sum of getCallingUid	0	0	0	0	0	0	1
Sum of getCallingPid	0	0	0	0	0	0	1
Sum of clearCallingIdentity	0	0	0	0	0	0	1
Sum of restoreCallingIdentity	0	0	0	0	0	0	1
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionSig	0	0	0	0	0	0	4
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	11
DevicePolicyManagerService							
Count of METHOD	49	55	116	116	116	118	142
Sum of getCallingUid	47	47	57	57	57	59	43
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	9	9	41	41	41	56	65
Sum of restoreCallingIdentity	9	9	41	41	41	56	65
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	47	50	57	57	57	59	43
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	1	4	25	25	25	23	53
DeviceStorageMonitorService							
Count of METHOD	45	47	0	0	0	0	0
Sum of getCallingUid	3	3	0	0	0	0	0
Sum of getCallingPid	1	1	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	42	43	0	0	0	0	0
DeviceStorageMonitorService$CachePackageDataObserver							
Count of METHOD	0	0	1	1	1	1	1
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	1
DiskStatsService							
Count of METHOD	29	32	33	33	33	33	34
Sum of getCallingUid	1	1	1	1	1	1	1
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	27	29	30	30	30	30	31
DisplayManagerService							
Count of METHOD	9	14	0	0	0	0	0
Sum of getCallingUid	0	3	0	0	0	0	0
Sum of getCallingPid	1	3	0	0	0	0	0
Sum of clearCallingIdentity	6	13	0	0	0	0	0
Sum of restoreCallingIdentity	6	13	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	8	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	2	0	0	0	0	0	0
DisplayManagerService$BinderService							
Count of METHOD	0	0	16	16	16	16	16
Sum of getCallingUid	0	0	3	3	3	3	3
Sum of getCallingPid	0	0	3	3	3	3	3
Sum of clearCallingIdentity	0	0	16	16	16	16	16
Sum of restoreCallingIdentity	0	0	16	16	16	16	16
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	7	7	7	7	7
Sum of permissionUndef	0	0	0	0	0	0	0

continued...

Feature counts for each System Service in AOSP baseline images (continued).

			Image				
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of None	0	0	0	0	0	0	0
DockObserver$BinderService							
Count of METHOD	0	0	33	33	33	33	34
Sum of getCallingUid	0	0	2	2	2	2	2
Sum of getCallingPid	0	0	1	1	1	1	1
Sum of clearCallingIdentity	0	0	1	1	1	1	1
Sum of restoreCallingIdentity	0	0	1	1	1	1	1
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	30	30	30	30	31
DreamManagerService							
Count of METHOD	8	8	0	0	0	0	0
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	7	7	0	0	0	0	0
Sum of clearCallingIdentity	7	7	0	0	0	0	0
Sum of restoreCallingIdentity	7	7	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	7	7	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
DreamManagerService$BinderService							
Count of METHOD	0	0	10	10	10	10	10
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	10	10	10	10	10
Sum of restoreCallingIdentity	0	0	10	10	10	10	10
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
DropBoxManagerService							
Count of METHOD	3	3	3	3	3	3	3
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	1	1	1	1	1	1
Sum of restoreCallingIdentity	0	1	1	1	1	1	1
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	2	1	1	1	1	1	1
EntropyMixer							
Count of METHOD	36	39	42	42	42	42	0
Sum of getCallingUid	1	1	1	1	1	1	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	35	37	40	40	40	40	0
FingerprintService$FingerprintServiceWrapper							
Count of METHOD	0	0	5	5	5	5	12
Sum of getCallingUid	0	0	0	0	0	0	1
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	3	3	3	3	6
Sum of permissionUndef	0	0	2	2	2	2	0
Sum of None	0	0	0	0	0	0	6
GraphicsStatsService							

continued...

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Count of METHOD	0	0	0	0	0	0	1
Sum of getCallingUid	0	0	0	0	0	0	1
Sum of getCallingPid	0	0	0	0	0	0	1
Sum of clearCallingIdentity	0	0	0	0	0	0	1
Sum of restoreCallingIdentity	0	0	0	0	0	0	1
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
HdmiControlService$BinderService							
Count of METHOD	0	0	32	32	32	33	33
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	32	32	32	33	33
InputManagerService							
Count of METHOD	15	15	17	17	17	17	17
Sum of getCallingUid	5	5	5	5	5	5	5
Sum of getCallingPid	6	6	6	6	6	6	6
Sum of clearCallingIdentity	1	1	0	0	0	0	0
Sum of restoreCallingIdentity	1	1	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	4	4	5	5	5	5	5
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	9	9	11	11	11	11	11
InputMethodManagerService							
Count of METHOD	29	30	36	36	36	36	43

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of getCallingUid	7	7	10	10	10	10	12
Sum of getCallingPid	0	0	0	0	0	0	2
Sum of clearCallingIdentity	11	11	11	11	11	11	11
Sum of restoreCallingIdentity	11	11	11	11	11	11	11
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	4	4	4	4	6
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	17	18	19	19	19	19	24
JobSchedulerService$JobSchedulerStub							
Count of METHOD	0	0	4	4	4	4	4
Sum of getCallingUid	0	0	4	4	4	4	4
Sum of getCallingPid	0	0	1	1	1	1	1
Sum of clearCallingIdentity	0	0	4	4	4	4	4
Sum of restoreCallingIdentity	0	0	4	4	4	4	4
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
LauncherAppsService$LauncherAppsImpl							
Count of METHOD	0	0	8	8	8	8	8
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	6	6	6	6	6
Sum of restoreCallingIdentity	0	0	6	6	6	6	6
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	2	2	2	2	2
LightsService							
Count of METHOD	3	4	0	0	0	0	0
Sum of getCallingUid	0	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	3	4	0	0	0	0	0
LocationManagerService							
Count of METHOD	28	28	32	32	32	32	33
Sum of getCallingUid	7	10	12	12	12	12	12
Sum of getCallingPid	2	2	2	2	2	2	6
Sum of clearCallingIdentity	13	14	16	16	16	16	16
Sum of restoreCallingIdentity	13	14	16	16	16	16	16
Sum of permissionNormal	1	1	1	1	1	1	1
Sum of permissionDangerous	8	8	8	8	8	8	0
Sum of permissionSig	1	2	2	2	2	2	2
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	9	8	10	10	10	10	14
LockSettingsService							
Count of METHOD	13	13	16	16	16	14	15
Sum of getCallingUid	8	3	3	3	3	3	3
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	3
Sum of restoreCallingIdentity	0	0	0	0	0	0	3
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	3	3	3	3	3	0
Sum of permissionSig	0	8	9	9	9	9	7
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	5	2	4	4	4	2	8
MediaProjectionManagerService$BinderService							
Count of METHOD	0	0	7	7	7	7	7
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

			Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of clearCallingIdentity	0	0	6	6	6	6	6
Sum of restoreCallingIdentity	0	0	6	6	6	6	6
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	6	6	6	6	6
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	1
MediaRouterService							
Count of METHOD	0	7	7	7	7	7	7
Sum of getCallingUid	0	1	1	1	1	1	1
Sum of getCallingPid	0	1	1	1	1	1	1
Sum of clearCallingIdentity	0	7	7	7	7	7	7
Sum of restoreCallingIdentity	0	7	7	7	7	7	7
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
MediaSessionService$SessionManagerImpl							
Count of METHOD	0	0	8	8	8	8	8
Sum of getCallingUid	0	0	6	6	6	6	6
Sum of getCallingPid	0	0	6	6	6	6	6
Sum of clearCallingIdentity	0	0	6	6	6	6	6
Sum of restoreCallingIdentity	0	0	6	6	6	6	6
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	2	2	2	2	2
MidiService							
Count of METHOD	0	0	0	0	0	0	11
Sum of getCallingUid	0	0	0	0	0	0	4
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	2

continued…

Feature counts for each System Service in AOSP baseline images (continued).

			Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of restoreCallingIdentity	0	0	0	0	0	0	2
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	6
MmsServiceBroker$BinderService							
Count of METHOD	0	0	16	16	16	14	14
Sum of getCallingUid	0	0	10	10	10	10	12
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	12	12	12	12	2
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	4	4	4	2	2
MountService							
Count of METHOD	36	37	43	43	45	45	63
Sum of getCallingUid	4	5	5	5	5	5	5
Sum of getCallingPid	1	1	1	1	1	1	0
Sum of clearCallingIdentity	0	0	0	0	0	0	4
Sum of restoreCallingIdentity	0	0	0	0	0	0	4
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	6	6	6	6	6	6	35
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	28	28	34	34	36	36	23
NetworkManagementService							
Count of METHOD	66	79	81	81	81	82	88
Sum of getCallingUid	6	6	6	6	6	6	9
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	59	73	64	64	64	65	69
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	1	0	11	11	11	11	11
NetworkPolicyManagerService							
Count of METHOD	13	13	16	16	16	16	17
Sum of getCallingUid	0	0	0	0	0	0	1
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	2	2	2	2	2	2	3
Sum of restoreCallingIdentity	2	2	2	2	2	2	3
Sum of permissionNormal	1	1	1	1	1	1	1
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	11	11	14	14	14	14	15
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	1	1	1	1	1	1	1
NetworkScoreService							
Count of METHOD	0	0	5	5	5	5	5
Sum of getCallingUid	0	0	3	3	3	3	3
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	4	4	4	4	4
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
NetworkStatsService							
Count of METHOD	8	8	8	8	8	9	10
Sum of getCallingUid	2	2	2	2	2	2	2
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	2	2	2	2	2	3	3
Sum of restoreCallingIdentity	2	2	2	2	2	3	3
Sum of permissionNormal	1	1	1	1	1	1	1

continued...

Feature counts for each System Service in AOSP baseline images (continued).

			Image				
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	6	6	6	6	6	7	6
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	1	1	1	1	1	1	3
NotificationManagerService							
Count of METHOD	9	16	0	0	0	0	0
Sum of getCallingUid	5	9	0	0	0	0	0
Sum of getCallingPid	4	4	0	0	0	0	0
Sum of clearCallingIdentity	2	4	0	0	0	0	0
Sum of restoreCallingIdentity	2	4	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	2	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	2	4	0	0	0	0	0
NotificationManagerService$5							
Count of METHOD	0	0	32	32	32	0	45
Sum of getCallingUid	0	0	8	8	8	0	14
Sum of getCallingPid	0	0	6	6	6	0	7
Sum of clearCallingIdentity	0	0	8	8	8	0	13
Sum of restoreCallingIdentity	0	0	8	8	8	0	13
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	10	10	10	0	7
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	12	12	12	0	19
NotificationManagerService$6							
Count of METHOD	0	0	0	0	0	33	0
Sum of getCallingUid	0	0	0	0	0	8	0
Sum of getCallingPid	0	0	0	0	0	6	0
Sum of clearCallingIdentity	0	0	0	0	0	8	0
Sum of restoreCallingIdentity	0	0	0	0	0	8	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0

continued...

Feature counts for each System Service in AOSP baseline images (continued).

Service				Image			
	AOSP-4.2.2	AOSP-4.4.4	AOSP-5.0	AOSP-5.0.1	AOSP-5.0.2	AOSP-5.1.1	AOSP-6.0
Sum of permissionSig	0	0	0	0	0	11	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	12	0
NotificationManagerService$StatusBarNotificationHolder							
Count of METHOD	0	0	1	1	1	1	1
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	1
NsdService							
Count of METHOD	2	2	2	2	2	2	2
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	1
Sum of permissionDangerous	1	1	1	1	1	1	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
PackageManagerService							
Count of METHOD	89	99	117	117	117	118	147
Sum of getCallingUid	30	33	39	39	39	39	54
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	3	6	6	6	6	6	11
Sum of restoreCallingIdentity	3	6	6	6	6	6	11
Sum of permissionNormal	1	1	1	1	1	1	1
Sum of permissionDangerous	2	2	2	2	2	2	0
Sum of permissionSig	30	33	37	37	37	37	54

continued...

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	49	55	65	65	65	66	72
PersistentDataBlockService$1							
Count of METHOD	0	0	7	7	7	7	7
Sum of getCallingUid	0	0	3	3	3	3	3
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	4	4	4	4	4
PowerManagerService							
Count of METHOD	22	24	0	0	0	0	0
Sum of getCallingUid	2	2	0	0	0	0	0
Sum of getCallingPid	2	2	0	0	0	0	0
Sum of clearCallingIdentity	17	17	0	0	0	0	0
Sum of restoreCallingIdentity	17	17	0	0	0	0	0
Sum of permissionNormal	2	2	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	12	12	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	5	7	0	0	0	0	0
PowerManagerService$BinderService							
Count of METHOD	0	0	22	22	22	22	24
Sum of getCallingUid	0	0	7	7	7	8	9
Sum of getCallingPid	0	0	2	2	2	2	2
Sum of clearCallingIdentity	0	0	19	19	19	19	21
Sum of restoreCallingIdentity	0	0	19	19	19	19	21
Sum of permissionNormal	0	0	2	2	2	2	1
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	14	14	14	15	15
Sum of permissionUndef	0	0	0	0	0	0	0

continued...

Feature counts for each System Service in AOSP baseline images (continued).

			Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of None	0	0	1	1	1	1	1
PrintManagerService							
Count of METHOD	0	17	0	0	0	0	0
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	16	0	0	0	0	0
Sum of restoreCallingIdentity	0	16	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	1	0	0	0	0	0
PrintManagerService$PrintManagerImpl							
Count of METHOD	0	0	17	17	17	17	17
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	16	16	16	16	16
Sum of restoreCallingIdentity	0	0	16	16	16	16	16
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	1
ProcessStatsService							
Count of METHOD	0	3	3	3	3	3	3
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	2	2	2	2	2	2
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	1	1	1	1	1	1

continued…

Feature counts for each System Service in AOSP baseline images (continued).

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
RestrictionsManagerService$RestrictionsManagerImpl							
Count of METHOD	0	0	5	5	5	5	5
Sum of getCallingUid	0	0	2	2	2	2	2
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	4	4	4	4	4
Sum of restoreCallingIdentity	0	0	4	4	4	4	4
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	1
SamplingProfilerService							
Count of METHOD	32	35	36	36	36	36	37
Sum of getCallingUid	1	1	1	1	1	1	1
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	30	32	33	33	33	33	34
SchedulingPolicyService							
Count of METHOD	1	1	1	1	1	1	1
Sum of getCallingUid	1	1	1	1	1	1	1
Sum of getCallingPid	1	1	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
SearchManagerService							

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Count of METHOD	7	7	8	8	8	8	8
Sum of getCallingUid	1	1	1	1	1	1	0
Sum of getCallingPid	0	1	1	1	1	1	0
Sum of clearCallingIdentity	2	1	2	2	2	2	2
Sum of restoreCallingIdentity	2	1	2	2	2	2	2
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	5	5	5	5	5	5	6
SerialService							
Count of METHOD	2	2	2	2	2	2	2
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	2	2	2	2	2	2	2
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
SipService							
Count of METHOD	9	0	0	0	0	0	0
Sum of getCallingUid	3	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	9	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
StatusBarManagerService							
Count of METHOD	21	22	26	26	26	28	35

continued...

Feature counts for each System Service in AOSP baseline images (continued).

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of getCallingUid	0	0	4	4	4	5	5
Sum of getCallingPid	0	0	4	4	4	5	5
Sum of clearCallingIdentity	0	0	8	8	8	10	10
Sum of restoreCallingIdentity	0	0	8	8	8	10	10
Sum of permissionNormal	3	3	3	3	3	3	3
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	12	12	15	15	15	17	19
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	6	7	8	8	8	8	13
TelephonyRegistry							
Count of METHOD	12	12	28	28	28	32	33
Sum of getCallingUid	11	11	18	18	18	18	20
Sum of getCallingPid	11	11	18	18	18	18	18
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	1	1	0	0	0	1	0
Sum of permissionSig	11	11	19	19	19	19	2
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	9	9	9	12	13
TextServicesManagerService							
Count of METHOD	9	9	9	9	9	9	9
Sum of getCallingUid	1	1	1	1	1	1	1
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	1	1	1	1	1	1	1
Sum of restoreCallingIdentity	1	1	1	1	1	1	1
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	3	3	3	3	3	3	3
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	5	5	5	5	5	5	5
ThrottleService							
Count of METHOD	7	0	0	0	0	0	0
Sum of getCallingUid	0	0	0	0	0	0	0

continued...

Feature counts for each System Service in AOSP baseline images (continued).

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	7	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
Trampoline							
Count of METHOD	0	0	0	0	0	27	28
Sum of getCallingUid	0	0	0	0	0	1	1
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	26	27
TrustManagerService$1							
Count of METHOD	0	0	5	5	5	7	9
Sum of getCallingUid	0	0	0	0	0	1	2
Sum of getCallingPid	0	0	0	0	0	1	2
Sum of clearCallingIdentity	0	0	0	0	0	0	2
Sum of restoreCallingIdentity	0	0	0	0	0	0	2
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	5	5	5	6	7
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
TvInputManagerService$BinderService							
Count of METHOD	0	0	31	31	31	32	39
Sum of getCallingUid	0	0	30	30	30	31	36
Sum of getCallingPid	0	0	30	30	30	31	36

continued...

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of clearCallingIdentity	0	0	31	31	31	32	39
Sum of restoreCallingIdentity	0	0	31	31	31	32	39
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	5	5	5	5	7
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
UiModeManagerService$5							
Count of METHOD	0	0	5	5	5	5	5
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	4	4	4	4	4
Sum of restoreCallingIdentity	0	0	4	4	4	4	4
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	1	1	1	1	1
UpdateLockService							
Count of METHOD	2	2	2	2	2	2	2
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	2	2	2	2	2	2	2
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
UsageStatsService							
Count of METHOD	5	5	0	0	0	0	0
Sum of getCallingUid	3	3	0	0	0	0	0
Sum of getCallingPid	3	3	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0

continued…

Feature counts for each System Service in AOSP baseline images (continued).

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	5	5	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
UsageStatsService$BinderService							
Count of METHOD	0	0	3	3	3	3	6
Sum of getCallingUid	0	0	0	0	0	0	3
Sum of getCallingPid	0	0	0	0	0	0	2
Sum of clearCallingIdentity	0	0	3	3	3	3	5
Sum of restoreCallingIdentity	0	0	3	3	3	3	5
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
UsbService							
Count of METHOD	18	19	19	19	19	19	23
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	3
Sum of restoreCallingIdentity	0	0	0	0	0	0	3
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	10	11	11	11	11	11	15
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	8	8	8	8	8	8	8
UserManagerService							
Count of METHOD	11	25	28	28	28	28	33
Sum of getCallingUid	8	17	21	21	21	21	25
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	1	2	5	5	5	5	4
Sum of restoreCallingIdentity	1	2	5	5	5	5	4

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	2	1	1	1	1	3
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	3	8	7	7	7	7	7
VibratorService							
Count of METHOD	4	4	4	4	4	4	4
Sum of getCallingUid	2	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	2	3	3	3	3	3	3
Sum of restoreCallingIdentity	2	3	3	3	3	3	3
Sum of permissionNormal	3	3	3	3	3	3	3
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	1	1	1	1	1	1	1
VoiceInteractionManagerService$VoiceInteractionManagerServiceStub							
Count of METHOD	0	0	11	11	11	11	26
Sum of getCallingUid	0	0	4	4	4	4	4
Sum of getCallingPid	0	0	4	4	4	4	1
Sum of clearCallingIdentity	0	0	11	11	11	11	22
Sum of restoreCallingIdentity	0	0	11	11	11	11	22
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	3	3	3	3	11
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
WallpaperManagerService							
Count of METHOD	9	9	12	12	12	12	14
Sum of getCallingUid	1	1	3	3	3	3	4
Sum of getCallingPid	3	3	4	4	4	4	5
Sum of clearCallingIdentity	3	3	3	3	3	3	3
Sum of restoreCallingIdentity	3	3	3	3	3	3	3
Sum of permissionNormal	2	2	3	3	3	3	4

continued…

Feature counts for each System Service in AOSP baseline images (continued).

				Image			
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	1	1	1	1	1	1	1
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	4	4	4	4	4	4	4
WebViewUpdateService$BinderService							
Count of METHOD	0	0	2	2	2	2	2
Sum of getCallingUid	0	0	1	1	1	1	1
Sum of getCallingPid	0	0	1	1	1	1	1
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	0	0	0	0	0	0	0
Sum of permissionSig	0	0	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
WifiP2pService							
Count of METHOD	1	2	0	0	0	0	0
Sum of getCallingUid	0	0	0	0	0	0	0
Sum of getCallingPid	0	0	0	0	0	0	0
Sum of clearCallingIdentity	0	0	0	0	0	0	0
Sum of restoreCallingIdentity	0	0	0	0	0	0	0
Sum of permissionNormal	0	0	0	0	0	0	0
Sum of permissionDangerous	1	1	0	0	0	0	0
Sum of permissionSig	0	1	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	0	0	0	0	0	0	0
WifiService							
Count of METHOD	43	52	0	0	0	0	0
Sum of getCallingUid	5	9	0	0	0	0	0
Sum of getCallingPid	3	4	0	0	0	0	0
Sum of clearCallingIdentity	4	7	0	0	0	0	0
Sum of restoreCallingIdentity	4	7	0	0	0	0	0
Sum of permissionNormal	12	14	0	0	0	0	0
Sum of permissionDangerous	21	20	0	0	0	0	0

continued...

Feature counts for each System Service in AOSP baseline images (continued).

	Image						
Service	**AOSP-4.2.2**	**AOSP-4.4.4**	**AOSP-5.0**	**AOSP-5.0.1**	**AOSP-5.0.2**	**AOSP-5.1.1**	**AOSP-6.0**
Sum of permissionSig	5	9	0	0	0	0	0
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	5	9	0	0	0	0	0
WindowManagerService							
Count of METHOD	79	80	78	78	78	79	82
Sum of getCallingUid	40	39	36	36	36	36	39
Sum of getCallingPid	41	36	32	32	32	32	33
Sum of clearCallingIdentity	19	20	20	20	20	21	22
Sum of restoreCallingIdentity	19	20	20	20	20	21	22
Sum of permissionNormal	0	0	0	0	0	0	5
Sum of permissionDangerous	4	4	5	5	5	5	0
Sum of permissionSig	42	41	37	37	37	37	40
Sum of permissionUndef	0	0	0	0	0	0	0
Sum of None	30	28	28	28	28	28	29

Appendix F

Vendor System Service Pivot Analysis

This appendix contains the results of a pivot analysis of System Service feature vectors from various vendor images compared with AOSP baselines with a similar Android version. Blank columns indicate either that the service is not present in that particular version of AOSP, or that *FeatureExtraction* was unable to process the JAR/class for that image.

Table entries show the total number of AIDL methods added or modified by the vendor for each service shown on the row labels, compared to the corresponding AOSP baseline shown in the column labels.

Table F.1: *Number of added or modified AIDL methods in each vendor service compared with the corresponding AOSP baseline.*

Service	Image											
	BLU-4.2.2 vs. AOSP-4.2.2	CM-4.4.4 vs. AOSP-4.4.4	CM-5.1.1 vs. AOSP-5.1.1	FireOS-5.1 vs. AOSP-5.1.1	LG-4.4.2 vs. AOSP-4.4.4	LG-5.0.2 vs. AOSP-5.0.2	LG-5.1 vs. AOSP-5.1.1	MotoX-5.0 vs. AOSP-5.0	SamsungEdge-5.1.1 vs. AOSP-5.1.1	SamsungNote8-4.4.2 vs. AOSP-4.4.4	SamsungS4-5.0.1 vs. AOSP-5.0.1	Xiaomi-4.4.4 vs. AOSP-4.4.4
ABTPersistenceService				72					72	72	72	
AccessibilityManagerService				16					20	8	16	
AccountManagerService				3	1				3	14	3	
ActivityManagerService		11	19	44	11	63	84		40	59	45	5
ActivityManagerService$CpuBinder									1	38		
ActivityManagerService$DbBinder									1	23		
ActivityManagerService$GmemBinder					31	32	32					
ActivityManagerService$GraphicsBinder									1	23		
ActivityManagerService$MemBinder									1	38		
ActivityManagerService$PermissionController										52		
ActivityManagerService$ProcessInfoService			1									
AlarmManagerService					1					2		
AlarmManagerService$2				3		1	1		3		3	
AlarmManagerService$3			6									
AlarmManagerServiceExt$SyncScheduler$TrafficAnalyzer$2				3					3		3	
AppDisablerService				1					1		1	
AppOpsService		3		2		1	1		1		1	2
AppWidgetService										1		
AppWidgetServiceImpl				3					4		2	
AudioService	7	2	8	34					28	16	26	4
BackupManagerService		2		26		1				5	4	18
BarBeamService				9					9	9	9	
BatteryService	17	1			1					31		
BatteryService$BinderService			1						1			
BatteryStatsService		3		1					1		1	
BluetoothA2dpService	15											
BluetoothManagerService		5	4	13	3	6	6		18	12	13	4
BluetoothProfileManagerService	7											
BluetoothSecureManagerService				13					13	13	13	
BluetoothService	88											
BluetoothSocketService	18											
CCModeService						3	3					
ClipboardExService				25					27		25	
ClipboardService				1		3	3		1		1	
CocktailBarManagerServiceContainer									59			
CommonTimeManagementService				1		1	1		2	1		
ConnectivityService	13		2	48	2	32	35		35	32	34	
ContentService				1					1	7	1	

continued...

Number of added or modified AIDL methods in each vendor service compared with the corresponding AOSP baseline (continued).

Service	**Image**											
	BLU-4.2.2 vs. AOSP-4.2.2	**CM-4.4.4 vs. AOSP-4.4.4**	**CM-5.1.1 vs. AOSP-5.1.1**	**FireOS-5.1 vs. AOSP-5.1.1**	**LG-4.4.2 vs. AOSP-4.4.4**	**LG-5.0.2 vs. AOSP-5.0.2**	**LG-5.1 vs. AOSP-5.1.1**	**MotoX-5.0 vs. AOSP-5.0**	**SamsungEdge-5.1.1 vs. AOSP-5.1.1**	**SamsungNote8-4.4.2 vs. AOSP-4.4.4**	**SamsungS4-5.0.1 vs. AOSP-5.0.1**	**Xiaomi-4.4.4 vs. AOSP-4.4.4**
ContextAwareService				17					19	17	17	
CoverManagerService				15					24	4	15	
DataSchedulerService						3	3					
DeviceManager3LMService					61	94	94					
DevicePolicyManagerService		1	1	73	2				70	49	56	
DeviceStorageMonitorService	22	3								9		
DirEncryptService				14					15	16	14	15
DiskStatsService									1	1		
DisplayManagerService					5					22		
DisplayManagerService$BinderService				26		4	4		27		26	
DockObserver$BinderService									1			
DreamManagerService	1											
DreamManagerService$BinderService			3									
EdgeGestureService		2	2									
EdgeManagerServiceImpl									4			
EmailKeystoreService									4			
EnterpriseContainerService										128		
EntropyMixer									1	1		
FastDownloadService						19	21					
FingerprintService$FingerprintServiceWrapper			11									
FMRadioService				97					97		97	
GestureService		2	2									
GlanceCardManagerService										14		
HarmonyEASService				8					8	8	8	
InjectionManagerService									5			
InputManagerService	1		1	21		1	1		24	26	21	
InputMethodManagerService	2	1		10	1	1	2		12	8	10	
InternalClipboardExService										23		
KiesUsbObserver				40					41	39	40	
KillSwitchService		7										
KtUcaService				25	25	25	25		25		25	
LGEncryptionService						11	11					
LightsService										6		
LocationManagerService	2			6					4		4	1
LockSettingsService		1		48					44	35	43	1
LpnetManagerService									12			
LSManager									3			
MediaSessionService$SessionManagerImpl				1							1	

continued...

Number of added or modified AIDL methods in each vendor service compared with the corresponding AOSP baseline (continued).

Service	Image											
	BLU-4.2.2 vs. AOSP-4.2.2	CM-4.4.4 vs. AOSP-4.4.4	CM-5.1.1 vs. AOSP-5.1.1	FireOS-5.1 vs. AOSP-5.1.1	LG-4.4.2 vs. AOSP-4.4.4	LG-5.0.2 vs. AOSP-5.0.2	LG-5.1 vs. AOSP-5.1.1	MotoX-5.0 vs. AOSP-5.0	SamsungEdge-5.1.1 vs. AOSP-5.1.1	SamsungNote8-4.4.2 vs. AOSP-4.4.4	SamsungS4-5.0.1 vs. AOSP-5.0.1	Xiaomi-4.4.4 vs. AOSP-4.4.4
MHPService						126	126					
MmsServiceBroker$BinderService				6								
MotionRecognitionService										9		
MountService	5	1	1	22	2				22	20	22	2
MSimTelephonyRegistry		12			12					12		
MultiWindowFacadeService										91		
MultiWindowFacadeService$BinderService				49					56		49	
NetworkManagementService	19	5	6	42	36	39	39	8	49	28	40	5
NetworkPolicyManagerService	3		1	3			1	1	3	3	4	
NetworkScoreService				4								
NetworkStatsService				2	1				2	2	2	
NotificationManagerService										5		
NotificationManagerService$10									1			
NotificationManagerService$5				34							3	
NotificationManagerService$6			2						6			
NotificationManagerService$8				1							1	
OemExtendedApi3LMService					9	9	9					
PackageManagerService	4	9	8	20		8		1	49	31	20	26
PalmMotionService										1		
PaymentManagerService									3			
PersistentDataBlockService$1				2					1			
PersonaFileManagerService										6		
PersonaManagerService				122					141	77	122	
PersonaPolicyManagerService				53					53	52	53	
PersonaStateManagerService										3		
PluginManagerService$PluginBinder				13					27		13	
PowerManagerService	1	9								43		4
PowerManagerService$BinderService			11	23		3	4	5	26		22	
PowerSaving3LMService						6	6					
ProcessStatsService										22		
ProfileManagerService		22										
QuickConnectService				3					3		3	
RCPManagerService				36					36	36	36	35
ReactiveService				8					9		8	
SAccessoryManager									2			
SamplingProfilerService									1	1		
ScepKeystoreProxyService				7					7		7	
SContextService				11					12	11	11	

continued...

Number of added or modified AIDL methods in each vendor service compared with the corresponding AOSP baseline (continued).

Service	Image											
	BLU-4.2.2 vs. AOSP-4.2.2	CM-4.4.4 vs. AOSP-4.4.4	CM-5.1.1 vs. AOSP-5.1.1	FireOS-5.1 vs. AOSP-5.1.1	LG-4.4.2 vs. AOSP-4.4.4	LG-5.0.2 vs. AOSP-5.0.2	LG-5.1 vs. AOSP-5.1.1	MotoX-5.0 vs. AOSP-5.0	SamsungEdge-5.1.1 vs. AOSP-5.1.1	SamsungNote8-4.4.2 vs. AOSP-4.4.4	SamsungS4-5.0.1 vs. AOSP-5.0.1	Xiaomi-4.4.4 vs. AOSP-4.4.4
SdpManagerService				13					14		13	
SearchManagerService				1					1	1	1	
SmartCoverService						4	5					
SpellManagerService										14		14
SpenGestureManagerService				12					13	9	12	9
StatusBarManagerService	14			12				2	15	20	10	1
TactileAssistService									8			
TelephonyRegistry	2			17	3	1	3		4	3	4	24
TextServicesManagerService				1								
ThemeIconManagerService					4	4	4					
ThemeService		11	12									
TimaService				50					50	27	50	
TorchService		4	8									
Trampoline			2						1			
TrustManagerService$1									2			
TwToolBoxService				6					6		6	
UsageStatsService										3		
UsageStatsService$BinderService				3					4		3	
UsbService				7	1				10	7	7	
UserManagerService		5		4	1				8	9	4	
VibratorService			2	9					10	14	9	
VoIPInterfaceManager				67					67	67	67	
VRManagerService									30			
VzwConnectivityService								3				
VzwLocationManagerService					16	16	16					
WallpaperManagerService		3	3	2					3	2	2	
WiFiAggregationService						6	6					
WiFiOffloadingService						28	28					
WifiP2pService	2									1		
WifiService	19	4			27					50		1
WindowManagerService		4	5	31	16	15	15	4	56	23	24	2
LocationPolicyManagerService												13
MiuiInitServer												8
MiuiUsbService												4
SecurityManagerService												14
Grand Total	**262**	**133**	**122**	**1416**	**272**	**566**	**589**	**24**	**1598**	**1565**	**1278**	**212**

Appendix G

LG-5.0.2 Test Results

Table G.1: Method-level test results for LG-5.0.2.

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*resetPacketDrop()*	*CHANGE_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(9905): Invoking test method... D/ServiceApiTest::SS(9905): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(9905): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(9905): Found method = public void android. os. INetworkManagementService $Stub $Proxy. resetPacketDrop() throws android. os. RemoteException I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/iptables -F OUTPUT) res=0 I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter W/libprocessgroup(960): failed to open /acct/uid_10161/pid_26958/cgroup. procs: No such file or directory D/KeyguardUpdateMonitor(1546): Intent. ACTION_BATTERYEX_CHANGED EXTRA_CHARGING_CURRENT : 1 / EXTRA_HVDCP_TYPE : false I/[SystemUI]LGPowerUI(1546): onReceive = com. lge. android. intent. action. BATTERYEX I/[SystemUI]LGPowerUI(1546): levelEx = 100, plugTypeEx = 2, WirelessAlignment = 0, chargingCurrent = 1, orientation = PORTRAIT I/[SystemUI]LGPowerUI(1546): On Skip Timer : true D/PowerService(2029): isFastChargerType: false, isCharging: false, isFastChargerConnected: false D/LIA_Service_NativeEventReceiver(2212): onReceive action : android. intent. action. PACKAGE_REMOVED = package:com. ratazzi. serviceapitest I/LIA_Service_PlatformUtil(2212): startLIAService() : Trying to start LIA service ... D/LIA_Service_LIAService(2212): onStartCommand() : LIAService started! - id :20, intent : Intent act=com. lge. ia pkg=com. lge. ia (has extras) I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/iptables -F INPUT) res=0 D/ServiceApiTest::MA(9905): ... finished.`	

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*runShellCommand()*	*CHANGE_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(26365): Invoking test method... D/ServiceApiTest::SS(26365): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(26365): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(26365): Found method = public void android. os. INetworkManagementService $Stub $Proxy. runShellCommand(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(960): LGE_DATA runShellCommand start cmd: lgdata_pulllog iptables -L D/NetworkManagementService(960): LGE_DATA runShellCommand done E/ServiceApiTest::MA(26365): java. lang. reflect. InvocationTargetException W/System. err(26365): java. lang. reflect. InvocationTargetException W/System. err(26365): at java. lang. reflect. Method. invoke(Native Method) W/System. err(26365): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(26365): at com. ratazzi. serviceapitest. NetworkManagementService. testMethodCall(NetworkManagementService. java:38) W/System. err(26365): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:20) W/System. err(26365): at android. app. Activity. performCreate(Activity. java:6020) W/System. err(26365): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1105) W/System. err(26365): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2284) W/System. err(26365): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2393) W/System. err(26365): at android. app. ActivityThread. access$800(ActivityThread. java:151) W/System. err(26365): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1309) W/System. err(26365): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(26365): at android. os. Looper. loop(Looper. java:135) W/System. err(26365): at android. app. ActivityThread. main(ActivityThread. java:5351) W/System. err(26365): at java. lang. reflect. Method. invoke(Native Method) W/System. err(26365): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(26365): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:908) W/System. err(26365): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:703) W/System. err(26365): Caused by: java. lang. IllegalArgumentException: Arguments must be separate from command W/System. err(26365): at android. os. Parcel. readException(Parcel. java:1544) W/System. err(26365): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(26365): at android. os. INetworkManagementService $Stub $Proxy. runShellCommand(INetworkManagementService. java:2479) W/System. err(26365): ... 17 more`	Must bypass this logic to cause execution: if (((!paramString. contains("iptables")) || (!paramString. contains("-L"))) && (!paramString. equals("lgdata_pulllog"))). Not seeing return message due to exception at target. Possible typo 'RunCommnad' [sic] in method, so maybe it doesn't work anyway.

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*SKTCatsPortForwarding()*	*CHANGE_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(20092): Invoking test method... D/ServiceApiTest::SS(20092): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(20092): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(20092): Found method = public void android. os. INetworkManagementService $Stub $Proxy. SKTCatsPortForwarding(java. lang. String) throws android. os. RemoteException V/NatController(328): [LGE_DATA] SKTCatsPortForwarding NatController : new_alias I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/iptables -I PREROUTING -t nat -d 192. 168. 1. 1 -j DNAT -to new_alias) res=0 I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/iptables -I FORWARD -d new_alias -j ACCEPT) res=0 D/CommandListener(328): [LGE_DATA] SKTCatsPortForwarding CommandListener D/ServiceApiTest::MA(20092): ... finished.`	
NetworkManagementService	*acceptPacket()*	*CHANGE_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(13240): Invoking test method... D/ServiceApiTest::SS(13240): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(13240): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(13240): Found method = public void android. os. INetworkManagementService $Stub $Proxy. acceptPacket(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(960): acceptPacket if only flag true and now is_dropping_packet = D/ServiceApiTest::MA(13240): ... finished.`	key net. is_dropping_packet unknown in running kernel

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
	addUpstreamV6Interface	*ACCESS_NETWORK_STATE*	Y	D/ServiceApiTest::MA(877): Invoking test method... D/ServiceApiTest::SS(877): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(877): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(877): Found method = public void android. os. INetworkManagementService $Stub $Proxy. addUpstreamV6Interface(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(960): addUpstreamInterface(abc1) D/CommandListener(328): TetherCmd::runCommand. argc: 4. argv[0]: tether D/CommandListener(328): command tether interface add_upstream abc1 D/TetherController(328): addUpstreamInterface(abc1) D/TetherController(328): . D/TetherController(328): int TetherController::getIfaceIndexForIface(const char *)() File path is /sys/class/net/abc0/ifindex E/TetherController(328): int TetherController::getIfaceIndexForIface(const char *)() Cannot read file : path /sys/class/net/abc0/ifindex, error No such file or directory D/TetherController(328): int TetherController::configureV6RtrAdv(): Upstream Iface: abc0 iface index: -1 D/TetherController(328): int TetherController::getIfaceIndexForIface(const char *)() File path is /sys/class/net/abc1/ifindex E/TetherController(328): int TetherController::getIfaceIndexForIface(const char *)() Cannot read file : path /sys/class/net/abc1/ifindex, error No such file or directory D/TetherController(328): int TetherController::configureV6RtrAdv(): Upstream Iface: abc1 iface index: -1 D/TetherController(328): Router advertisement daemon already stopped D/TetherController(328): Router advertisement daemon running D/radish (934): radish_bridge_debug_cmds: ebtables -t broute -L D/radish (934): Bridge table: broute D/radish (934): D/radish (934): Bridge chain: BROUTING, entries: 0, policy: ACCEPT D/radish (934): radish_bridge_debug_cmds: brctl show D/radish (934): bridge namebridge idSTP enabledinterfaces D/radish (934): radish_bridge_debug_cmds: netstat -ap ... D/radish (934): radish_init_bridge: brctl addbr bridge0 I/Netd (328): M: Get netlink event, ifname: bridge0 action: 1 D/wpa_supplicant(4333): nl80211: Ignore RTM_NEWLINK event for foreign ifindex 23 D/radish (934): radish_init_bridge: echo 1 > /proc/sys/net/ipv6/conf/bridge0/optimistic_dad D/radish (934): radish_init_bridge: echo 10 > /proc/sys/net/ipv6/neigh/bridge0/retrans_time_ms I/Netd (328): M: Get netlink event, ifname: bridge0 action: 5 D/radish (934): radish_init_bridge: ifconfig bridge0 up D/wpa_supplicant(4333): nl80211: Ignore RTM_NEWLINK event for foreign ifindex 23 I/Netd (328): M: Get netlink event, ifname: bridge0 action: 4 I/Netd (328): M: Get netlink event, ifname: bridge0 action: 9 D/radish (934): radish_init_bridge: ebtables -t broute -A BROUTING -p ipv4 -j DROP D/libc-netbsd(960): [getaddrinfo]: hostname=xxxxx; servname=(null); cache_mode=(null), netid=0; mark=0 D/libc-netbsd(960): [getaddrinfo]: ai_addrlen=0; ai_canonname=xxxxx; ai_flags=4; ai_family=0 I/Netd (328): M: Get netlink event, ifname: bridge0 action: 6 D/libc-netbsd(960): [getaddrinfo]: hostname=xxxxx; servname=(null); cache_mode=(null), netid=0; mark=0 D/libc-netbsd(960): [getaddrinfo]: ai_addrlen=0; ai_canonname=xxxxx; ai_flags=4; ai_family=0 ... D/ServiceApiTest::MA(877): ... finished.	Interface added abc0, abc1

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*appendRouteWithMetric()*	*CONNECTIVITY_INTERNAL*	N	`D/ServiceApiTest::MA(21381): Invoking test method... D/ServiceApiTest::SS(21381): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(21381): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(21381): Found method = public boolean android. os. INetworkManagementService $Stub $Proxy. appendRouteWithMetric(java. lang. String,int,android. net. RouteInfo) throws android. os. RemoteException E/ServiceApiTest::MA(21381): java. lang. reflect. InvocationTargetException W/System. err(21381): java. lang. reflect. InvocationTargetException W/System. err(21381): at java. lang. reflect. Method. invoke(Native Method) W/System. err(21381): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(21381): at com. ratazzi. serviceapitest. NetworkManagementService. testMethodCall(NetworkManagementService. java:41) W/System. err(21381): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:20) W/System. err(21381): at android. app. Activity. performCreate(Activity. java:6020) W/System. err(21381): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1105) W/System. err(21381): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2284) W/System. err(21381): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2393) W/System. err(21381): at android. app. ActivityThread. access$800(ActivityThread. java:151) W/System. err(21381): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1309) W/System. err(21381): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(21381): at android. os. Looper. loop(Looper. java:135) W/System. err(21381): at android. app. ActivityThread. main(ActivityThread. java:5351) W/System. err(21381): at java. lang. reflect. Method. invoke(Native Method) W/System. err(21381): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(21381): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:908) W/System. err(21381): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:703) W/System. err(21381): Caused by: java. lang. SecurityException: NetworkManagementService: Neither user 10164 nor current process has android. permission. CONNECTIVITY_INTERNAL. W/System. err(21381): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(21381): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(21381): at android. os. INetworkManagementService $Stub $Proxy. appendRouteWithMetric(INetworkManagementService. java:3265) W/System. err(21381): ... 17 more`	System permission
NetworkManagementService	*clearRouteIpv6()*	*CONNECTIVITY_INTERNAL*	N		System permission
NetworkManagementService	*clearSourceRouteIpv6()*	*CONNECTIVITY_INTERNAL*	N		System permission
NetworkManagementService	*disableNatIpv6()*	*CONNECTIVITY_INTERNAL*	N		System permission

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*dropPacket()*	*CHANGE_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(27882): Invoking test method... D/ServiceApiTest::SS(27882): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(27882): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(27882): Found method = public void android. os. INetworkManagementService $Stub $Proxy. dropPacket(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(960): drop Packet if only flag false and now is_dropping_packet = D/ServiceApiTest::MA(27882): ... finished.`	key net. is_dropping_packet unknown in running kernel
NetworkManagementService	*enableNatIpv6()*	*CONNECTIVITY_INTERNAL*	N		System permission

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*getInterfaceForEps()*	*None*	Y	`D/ServiceApiTest::MA(30244): Invoking test method... D/ServiceApiTest::SS(30244): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(30244): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(30244): Found method = public java. lang. String android. os. INetworkManagementService $Stub $Proxy. getInterfaceForEps() throws android. os. RemoteException E/ServiceApiTest::MA(30244): java. lang. reflect. InvocationTargetException W/System. err(30244): java. lang. reflect. InvocationTargetException W/System. err(30244): at java. lang. reflect. Method. invoke(Native Method) W/System. err(30244): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(30244): at com. ratazzi. serviceapitest. NetworkManagementService. testMethodCall(NetworkManagementService. java:41) W/System. err(30244): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:20) W/System. err(30244): at android. app. Activity. performCreate(Activity. java:6020) W/System. err(30244): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1105) W/System. err(30244): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2284) W/System. err(30244): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2393) W/System. err(30244): at android. app. ActivityThread. access$800(ActivityThread. java:151) W/System. err(30244): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1309) W/System. err(30244): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(30244): at android. os. Looper. loop(Looper. java:135) W/System. err(30244): at android. app. ActivityThread. main(ActivityThread. java:5351) W/System. err(30244): at java. lang. reflect. Method. invoke(Native Method) W/System. err(30244): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(30244): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:908) W/System. err(30244): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:703) W/System. err(30244): Caused by: java. lang. SecurityException: Only available to AID_SYSTEM W/System. err(30244): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(30244): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(30244): at android. os. INetworkManagementService $Stub $Proxy. getInterfaceForEps(INetworkManagementService. java:2692) W/System. err(30244): ... 17 more`	Requires uid==1000
NetworkManagementService	*getIpv6ForwardingEnabled()*	*CONNECTIVITY_INTERNAL*	N		System permission
NetworkManagementService	*getMtu()*	*CONNECTIVITY_INTERNAL*	N		System permission
NetworkManagementService	*getNetworkStatsUidInterface()*	*CONNECTIVITY_INTERNAL*	N		System permission

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*getRouteList_debug()*	*None*	Y	D/ServiceApiTest::SS(4348): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(4348): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(4348): Found method = public void android. os. INetworkManagementService $Stub $Proxy. getRouteList_debug(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(960): [LGE_DATA_DEBUG] route dest :00000000 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route gate :0914A8C0 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route flags :0003 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route MTU :0 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route dest :0014A8C0 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route gate :00000000 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route flags :0001 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route MTU :0 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route dest :0014A8C0 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route gate :00000000 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route flags :0001 D/NetworkManagementService(960): [LGE_DATA_DEBUG] route MTU :0 D/ServiceApiTest::MA(4348): ... finished.	
NetworkManagementService	*isExceptionalUidForEps()*	*None*	Y		Requires uid==1000
NetworkManagementService	*isFirewallForEpsEnabled()*	*None*	Y		Requires uid==1000
NetworkManagementService	*packetList_InDrop()*	*CHANGE_NETWORK_STATE*	Y	D/ServiceApiTest::MA(10031): Invoking test method... D/ServiceApiTest::SS(10031): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(10031): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(10031): Found method = public boolean android. os. INetworkManagementService $Stub $Proxy. packetList_Indrop() throws android. os. RemoteException I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter E/logwrapper(10083): executing /system/bin/iptables_ds-save failed: No such file or directory I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/iptables_ds-save) res=0 I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/chmod 777 /data/ip_tables_save_temp) res=0 D/NetworkManagementService(960): [LGE_DATA] Packetdrop list was lefted !!! () D/ServiceApiTest::MA(10031): ... finished.	

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	packetList_InDrop_view()	None	Y	D/ServiceApiTest::MA(14472): Invoking test method... D/ServiceApiTest::SS(14472): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(14472): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(14472): Found method = public void android. os. INetworkManagementService $Stub $Proxy. packetList_Indrop_view() throws android. os. RemoteException D/ServiceApiTest::MA(14472): ... finished.	Needs debug_interfaceName != null
NetworkManagementService	removeLegacyRouteForNetId()	CONNECTIVITY_INTERNAL	N		System permission
	registerObserverEx				
NetworkManagementService	removeUpstreamV6Interface()	ACCESS_NETWORK_STATE	Y	D/ServiceApiTest::MA(21122): Invoking test method... D/ServiceApiTest::SS(21122): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(21122): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(21122): Found method = public void android. os. INetworkManagementService $Stub $Proxy. removeUpstreamV6Interface(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(960): removeUpstreamInterface(abc0) D/CommandListener(328): TetherCmd::runCommand. argc: 4. argv[0]: tether D/TetherController(328): int TetherController::getIfaceIndexForIface(const char *)() File path is /sys/class/net/abc1/ifindex E/TetherController(328): int TetherController::getIfaceIndexForIface(const char *)() Cannot read file : path /sys/class/net/abc1/ifindex, error No such file or directory D/TetherController(328): int TetherController::configureV6RtrAdv(): Upstream Iface: abc1 iface index: -1 D/TetherController(328): Router advertisement daemon stopped D/TetherController(328): Need atleast two interfaces to start Router advertisement daemon D/ServiceApiTest::MA(21122): ... finished.	
NetworkManagementService	replaceDefaultRouteWithTable()	CONNECTIVITY_INTERNAL	N		System permission
NetworkManagementService	removeSrcRouteForNetId()con	CONNECTIVITY_INTERNAL	N		System permission
NetworkManagementService	resetConnectionsForEps()	None	Y		Requires uid==1000
NetworkManagementService	resetPacketDrop()	CHANGE_NETWORK_STATE	Y	D/ServiceApiTest::MA(8222): Invoking test method... D/ServiceApiTest::SS(8222): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(8222): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(8222): Found method = public void android. os. INetworkManagementService $Stub $Proxy. resetPacketDrop() throws android. os. RemoteException I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/iptables -F OUTPUT) res=0 I/NatController(328): [LG_DATA] NatController runIptablesCmd status : 1000 I/NatController(328): NatController android_fork_execvp enter I/NatController(328): NatController android_fork_execvp exit V/NatController(328): runCmd(/system/bin/iptables -F INPUT) res=0 D/ServiceApiTest::MA(8222): ... finished.	

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*setDhcpv6Enabled()*	*CHANGE_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(12557): Invoking test method... D/ServiceApiTest::SS(12557): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(12557): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(12557): Found method = public void android. os. INetworkManagementService $Stub $Proxy. setDhcpv6Enabled(boolean,java. lang. String) throws android. os. RemoteException W/CommandListener(328): IPv6 tethering feature is disabled, ingore this command add D/ServiceApiTest::MA(12557): ... finished.`	
NetworkManagementService	*setInterfaceAlias()*	*CHANGE_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(18386): Invoking test method... D/ServiceApiTest::SS(18386): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(18386): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(18386): Found method = public void android. os. INetworkManagementService $Stub $Proxy. setInterfaceAlias(java. lang. String,java. lang. String) throws android. os. RemoteException V/ (328): QcRouteController D/ServiceApiTest::MA(18386): ... finished.`	
NetworkManagementService	*setTcpWindowScaling()*			`D/ServiceApiTest::MA(22273): Invoking test method... D/ServiceApiTest::SS(22273): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(22273): Returned object = android. os. INetworkManagementService $Stub $Proxy@2aa9e56 D/ServiceApiTest::NMS(22273): Found method = public void android. os. INetworkManagementService $Stub $Proxy. setTcpWindowScaling(boolean) throws android. os. RemoteException D/ServiceApiTest::MA(22273): ... finished.`	
NetworkManagementService	*setWifiHighTrafficMode()*	*CONNECTIVITY_INTERNAL*	N		System permission
InputManagerService	*setCustomHoverIcon()*				
BackupManagerService	*clearBackupData()*	*BACKUP*	Y	`D/ServiceApiTest::MA(27192): Invoking test method... D/ServiceApiTest::SS(27192): Returned object = android. app. backup. IBackupManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::BMS(27192): Returned object = android. app. backup. IBackupManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::BMS(27192): Found method = public void android. app. backup. IBackupManager $Stub $Proxy. clearBackupData(java. lang. String,java. lang. String) throws android. os. RemoteException V/BackupManagerService(960): clearBackupData() of com. ratazzi. serviceapitest on android/com. android. internal. backup. LocalTransport E/ServiceApiTest::MA(27192): java. lang. reflect. InvocationTargetException`	System permission âĂŞ doesn't throw security exception, but instead throws NPE because hashset apps. contains(packageName) is null. But âĂIJPrivileged caller... âĂİ log is not reached and neither is âĂIJFound the app... âĂİ
AlarmManagerService	*updateBlockedUids()*	*None*	Y	`D/ServiceApiTest::MA(26145): Invoking test method... D/ServiceApiTest::SS(26145): Returned object = android. app. IAlarmManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::AMS(26145): Returned object = android. app. IAlarmManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::AMS(26145): Found method = public void android. app. IAlarmManager $Stub $Proxy. updateBlockedUids(int,boolean) throws android. os. RemoteException D/ServiceApiTest::MA(26145): ... finished.`	Requires uid==1000

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
CCModeService	*cc_mode_disable(),* *cc_mode_enable(),* *cc_mode_isSupported()*	*ACCESS_CC_MODE*	Y	`D/ServiceApiTest::MA(12694): Invoking test method... D/ServiceApiTest::SS(12694): Returned object = android. security. ICCModeService $Stub $Proxy@2aa9e56 D/ServiceApiTest::CCMS(12694): Returned object = android. security. ICCModeService $Stub $Proxy@2aa9e56 D/ServiceApiTest::CCMS(12694): Found method = public int android. security. ICCModeService $Stub $Proxy. cc_mode_disable() throws android. os. RemoteException E/CC_MODE (960): Permission Denial: setSystemProperty() from pid=12694, uid=10166 requires com. lge. security. cc_mode_disable E/ServiceApiTest::MA(12694): java. lang. reflect. InvocationTargetException W/System. err(12694): java. lang. reflect. InvocationTargetException W/System. err(12694): at java. lang. reflect. Method. invoke(Native Method) W/System. err(12694): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(12694): at com. ratazzi. serviceapitest. CCModeService. testMethodCall(CCModeService. java:42) W/System. err(12694): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:24) W/System. err(12694): at android. app. Activity. performCreate(Activity. java:6020) W/System. err(12694): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1105) W/System. err(12694): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2284) W/System. err(12694): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2393) W/System. err(12694): at android. app. ActivityThread. access$800(ActivityThread. java:151) W/System. err(12694): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1309) W/System. err(12694): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(12694): at android. os. Looper. loop(Looper. java:135) W/System. err(12694): at android. app. ActivityThread. main(ActivityThread. java:5351) W/System. err(12694): at java. lang. reflect. Method. invoke(Native Method) W/System. err(12694): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(12694): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:908) W/System. err(12694): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:703) W/System. err(12694): Caused by: java. lang. SecurityException: Permission Denial: setSystemProperty() from pid=12694, uid=10166 requires com. lge. security. cc_mode_disable W/System. err(12694): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(12694): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(12694): at android. security. ICCModeService $Stub $Proxy. cc_mode_disable(ICCModeService. java:117) W/System. err(12694): ... 17 more`	Custom system permission: W/PackageManager(960): Not granting permission com. lge. permission. ACCESS_CC_MODE to package com. ratazzi. serviceapitest (protectionLevel=18 flags=0x48be46)

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ClipboardService	*getPrimaryClip()*	*None*	Y	`D/ServiceApiTest::MA(322): Invoking test method... D/ServiceApiTest::SS(322): Returned object = android. content. IClipboard $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(322): Returned object = android. content. IClipboard $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(322): Found method = public android. content. ClipData android. content. IClipboard $Stub $Proxy. getPrimaryClip(java. lang. String) throws android. os. RemoteException W/AppOps (960): Bad call: specified package com. lge. email under uid 10166 but it is really 10052 E/ServiceApiTest::MA(322): java. lang. reflect. InvocationTargetException W/System. err(322): java. lang. reflect. InvocationTargetException W/System. err(322): at java. lang. reflect. Method. invoke(Native Method) W/System. err(322): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(322): at com. ratazzi. serviceapitest. ClipboardService. testMethodCall(ClipboardService. java:42) W/System. err(322): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:24) W/System. err(322): at android. app. Activity. performCreate(Activity. java:6020) W/System. err(322): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1105) W/System. err(322): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2284) W/System. err(322): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2393) W/System. err(322): at android. app. ActivityThread. access$800(ActivityThread. java:151) W/System. err(322): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1309) W/System. err(322): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(322): at android. os. Looper. loop(Looper. java:135) W/System. err(322): at android. app. ActivityThread. main(ActivityThread. java:5351) W/System. err(322): at java. lang. reflect. Method. invoke(Native Method) W/System. err(322): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(322): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:908) W/System. err(322): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:703) W/System. err(322): Caused by: java. lang. SecurityException: com. lge. email from uid 10166 not allowed to perform READ_CLIPBOARD W/System. err(322): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(322): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(322): at android. content. IClipboard $Stub $Proxy. getPrimaryClip(IClipboard. java:187) W/System. err(322): ... 17 more`	AppOps performs calling vs. target uid comparison. LG changes are for 3LM.

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ConnectivityService	*getDebugInfo()*	*None*	Y	`D/ServiceApiTest::MA(11426): Invoking test method... D/ServiceApiTest::SS(11426): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(11426): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(11426): Found method = public [D android. net. IConnectivityManager $Stub $Proxy. getDebugInfo(int,int) throws android. os. RemoteException D/ConnectivityService(960): ePDG Feature change : Disable D/ServiceApiTest::MA(11426): ... finished.`	Able to toggle ePDGMode boolean used in many non-AIDL public and private methods. To enable, pass 0,1 âĂŞ to disable pass 0,0. passing 1,x calls ePDGTracker. getDebugInfo which returns null.
ConnectivityService	*checkLteConnectState()*	*None*	Y	`D/ServiceApiTest::MA(4020): Invoking test method... D/ServiceApiTest::SS(4020): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(4020): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(4020): Found method = public int android. net. IConnectivityManager $Stub $Proxy. checkLteConnectState() throws android. os. RemoteException D/ConnectivityService(960): checkLteConnectState() Entry ! D/ConnectivityService(960): [checkLteConnectState()] mLteState == -1 D/ConnectivityService(960): [checkLteConnectState()] DctConstants. State. IDLE == IDLE, DctConstants. State. RETRYING == RETRYING D/ConnectivityService(960): [checkLteConnectState()] mLteState is out of range !!! D/ServiceApiTest::MA(4020): ... finished.`	
ConnectivityService	*checkVzwNetType()*	*None*	Y	`D/ServiceApiTest::MA(26279): Invoking test method... D/ServiceApiTest::SS(26279): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(26279): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(26279): Found method = public int android. net. IConnectivityManager $Stub $Proxy. checkVzwNetType(int) throws android. os. RemoteException D/ConnectivityService(960): SecureSetting caller UID: 10166 D/ConnectivityService(960): SecureSetting pkg: com. ratazzi. serviceapitest D/ConnectivityService(960): SecureSetting oops didn't find a match D/ConnectivityService(960): mSystemImage : false mSignedFromVZW : false mContainVzwAppApn_MetaTag : true D/ConnectivityService(960): checkVzwNetType set from 19 to 5 D/ServiceApiTest::MA(26279): ... finished.`	Checks if caller is from system image, VZW signed or has VZW meta tag in manifest. Was only able to get a true for the last one by setting tag in manifest. Otherwise seems like just a getter.

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ConnectivityService	*getNetPrefer()*	*None*	Y	`D/ServiceApiTest::MA(21256): Invoking test method... D/ServiceApiTest::SS(21256): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(21256): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(21256): Found method = public int android. net. IConnectivityManager $Stub $Proxy. getNetPrefer(java. lang. String) throws android. os. RemoteException D/ConnectivityService(960): ePDG ConnectivityService getNetPrefer : D/ePDGTracker(960): createPDGConnection E D/ePDG (960): [ePDGDC-1] ePDGConnection constructor E D/ePDG (960): [ePDGDC-1] ePDGConnection constructor X D/ePDGTracker(960): createPDGConnection() X id=1 D/ePDG (960): [ePDGDC-1] Defatult state enter D/ePDG (960): [ePDGDC-1] clearSettings D/ePDG (960): [ePDGDC-1] DcNonetworkState state enter D/ePDG (960): [ePDGDC-1] DcNonetworkState state exit D/ePDG (960): [ePDGDC-1] DcReadyState state enter D/ePDGTracker(960): getNetPrefer:2 D/ServiceApiTest::MA(21256): ... finished.`	By passing parameter âĂIJMM-SâĂİ, able to invoke ePDG-Tracker. getNetPrefer which invokes private createPDGConnection
ConnectivityService	*getNetworkStatus_for_kt_kaf()*	*ACCESS_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(31658): Invoking test method... D/ServiceApiTest::SS(31658): Returned object = android. net. IConnectivityManager $Stub $Proxy@1dfe4ebd D/ServiceApiTest::CS(31658): Returned object = android. net. IConnectivityManager $Stub $Proxy@1dfe4ebd D/ServiceApiTest::CS(31658): Found method = public int android. net. IConnectivityManager $Stub $Proxy. getNetworkStatus_for_kt_kaf(int) throws android. os. RemoteException D/ConnectivityService(981): [LGE_DATA-KAF] getNetworkStatus_for_kt_kaf :: networkType=1 / nai. validated= true D/DataConnectionInterface (981): getDataOnRoamingEnabled false!! D/DataConnectionInterface (981): getDataOnRoamingEnabled false!! D/DATA (981): For KT Roaming isRoaming = 0isDataRoaming = false D/DataConnectionInterface (981): getDataOnRoamingEnabled false!! D/DATA (981): For KT Roaming isRoaming = 0isDataRoaming = false D/ServiceApiTest::CS(31658): Result = 0 D/ServiceApiTest::MA(31658): ... finished.`	Passed parameter "1" (TYPE_WIFI). Permission is enforced in called method.
ConnectivityService	*getPcscfAddress()*	*None*	Y	`D/ServiceApiTest::MA(17105): Invoking test method... D/ServiceApiTest::SS(17105): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(17105): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(17105): Found method = public [Ljava. lang. String; android. net. IConnectivityManager $Stub $Proxy. getPcscfAddress(java. lang. String,java. lang. String) throws android. os. RemoteException D/ServiceApiTest::CS(17105): Result = null D/ServiceApiTest::MA(17105): ... finished.`	Couldn't get any result other than null. Probably depends on active ePDG connection
ConnectivityService	*getTetheredIfacePairs()*	*ACCESS_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(22585): Invoking test method... D/ServiceApiTest::SS(22585): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(22585): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(22585): Found method = public [Ljava. lang. String; android. net. IConnectivityManager $Stub $Proxy. getTetheredIfacePairs() throws android. os. RemoteException D/ServiceApiTest::CS(22585): Result = H4sIAAAAAAAAAA== D/ServiceApiTest::MA(22585): ... finished.`	

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ConnectivityService	*mobileDataPdpReset()*	*None*	Y	`D/ServiceApiTest::MA(30778): Invoking test method... D/ServiceApiTest::SS(30778): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(30778): Returned object = android. net. IConnectivityManager $Stub $Proxy@2aa9e56 D/ServiceApiTest::CS(30778): Found method = public void android. net. IConnectivityManager $Stub $Proxy. mobileDataPdpReset() throws android. os. RemoteException D/ConnectivityService(960): [LGE_DATA] mobileDataPdpReset is not allowed when it is not in network roaming. D/ServiceApiTest::CS(30778): Result = null D/ServiceApiTest::MA(30778): ... finished.`	Can't put device in roaming, so cannot test fully.

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes

ConnectivityService	*setDataBlock()*	*None*	Y	`D/ServiceApiTest::MA( 9147): Invoking test method... D/ServiceApiTest::SS(`
				`9147): Returned object = android. net. IConnectivityManager $Stub $Proxy@24780b39`
				`D/ServiceApiTest::CS( 9147): Returned object = android. net. IConnectivityManager`
				`$Stub $Proxy@24780b39 D/ServiceApiTest::CS( 9147): Found method = public void`
				`android. net. IConnectivityManager $Stub $Proxy. setDataBlock(boolean) throws`
				`android. os. RemoteException D/ConnectivityService( 960): [setDataBlock()] enable`
				`= false D/PcoTracker( 960): dataBlock set to false D/PcoTracker( 960): dataBlock`
				`Unblocked I/BandwidthController( 328): [LG_DATA] BandwidthController runIptablesCmd`
				`status : 1000 I/BandwidthController( 328): BandwidthController android_fork_execvp`
				`enter I/BandwidthController( 328): BandwidthController android_fork_execvp exit`
				`I/BandwidthController( 328): [LG_DATA] BandwidthController runIptablesCmd status`
				`: 1000 I/BandwidthController( 328): BandwidthController android_fork_execvp`
				`enter I/BandwidthController( 328): BandwidthController android_fork_execvp exit`
				`D/ServiceApiTest::CS( 9147): Result = null D/ServiceApiTest::MA( 9147): ... finished.`
				`D/ServiceApiTest::MA(12606): Invoking test method... D/ServiceApiTest::SS(12606):`
				`Returned object = android. net. IConnectivityManager $Stub $Proxy@24780b39`
				`D/ServiceApiTest::CS(12606): Returned object = android. net. IConnectivityManager`
				`$Stub $Proxy@24780b39 D/ServiceApiTest::CS(12606): Found method = public void`
				`android. net. IConnectivityManager $Stub $Proxy. setDataBlock(boolean) throws`
				`android. os. RemoteException D/ConnectivityService( 960): [setDataBlock()] enable`
				`= true D/PcoTracker( 960): dataBlock set to true D/PcoTracker( 960): dataBlock`
				`Blocked I/BandwidthController( 328): [LG_DATA] BandwidthController runIptablesCmd`
				`status : 1000 I/BandwidthController( 328): BandwidthController android_fork_execvp`
				`enter I/BandwidthController( 328): BandwidthController android_fork_execvp exit`
				`I/BandwidthController( 328): [LG_DATA] BandwidthController runIptablesCmd status`
				`: 1000 I/BandwidthController( 328): BandwidthController android_fork_execvp`
				`enter I/BandwidthController( 328): BandwidthController android_fork_execvp exit`
				`I/BandwidthController( 328): [LG_DATA] BandwidthController runIptablesCmd status`
				`: 1000 I/BandwidthController( 328): BandwidthController android_fork_execvp`
				`enter I/BandwidthController( 328): BandwidthController android_fork_execvp exit`
				`I/BandwidthController( 328): [LG_DATA] BandwidthController runIptablesCmd status`
				`: 1000 I/BandwidthController( 328): BandwidthController android_fork_execvp`
				`enter I/BandwidthController( 328): BandwidthController android_fork_execvp exit`
				`D/ServiceApiTest::CS(12606): Result = null D/ServiceApiTest::MA(12606): ... finished.`
					Requires kernel config ro. build. characteristics=tablet. After forcing this condition, successfully called. Still looking for effect on device.

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ConnectivityService	*setLteMobileDataEnabled()*	*None*	Y	D/ServiceApiTest::MA(11945): Invoking test method... D/ServiceApiTest::SS(11945): Returned object = android. net. IConnectivityManager $Stub $Proxy@4591550 D/ServiceApiTest::CS(11945): Returned object = android. net. IConnectivityManager $Stub $Proxy@4591550 D/ServiceApiTest::CS(11945): Found method = public void android. net. IConnectivityManager $Stub $Proxy. setLteMobileDataEnabled(boolean) throws android. os. RemoteException D/ConnectivityService(969): [setLteMobileDataEnabled()] enable == true W/ContextImpl(969): Calling a method in the system process without a qualified user: android. app. ContextImpl. sendBroadcast:1433 com. android. server. ConnectivityService. setLteMobileDataEnabled:2117 android. net. IConnectivityManager$Stub. onTransact:987 android. os. Binder. execTransact:446 <bottom of call stack> W/libprocessgroup(969): failed to open /acct/uid_10161/pid_10748/cgroup. procs: No such file or directory D/ConnectivityService(969): [setLteMobileDataEnabled()] The intent of set_network_mode is sent now ! D/ConnectivityService(969): [setLteMobileDataEnabled()] The intent of lte_data_completed_action is sent now with connecting_4G ! W/ContextImpl(969): Calling a method in the system process without a qualified user: android. app. ContextImpl. sendBroadcast:1433 com. android. server. ConnectivityService. setLteMobileDataEnabled:2128 android. net. IConnectivityManager$Stub. onTransact:987 android. os. Binder. execTransact:446 <bottom of call stack> D/ServiceApiTest::CS(11945): Result = null D/ServiceApiTest::MA(11945): ... finished. D/ServiceApiTest::MA(12765): Invoking test method... D/ServiceApiTest::SS(12765): Returned object = android. net. IConnectivityManager $Stub $Proxy@12dc3a02 D/ServiceApiTest::CS(12765): Returned object = android. net. IConnectivityManager $Stub $Proxy@12dc3a02 D/ServiceApiTest::CS(12765): Found method = public void android. net. IConnectivityManager $Stub $Proxy. setLteMobileDataEnabled(boolean) throws android. os. RemoteException D/ConnectivityService(969): [setLteMobileDataEnabled()] enable == false W/ContextImpl(969): Calling a method in the system process without a qualified user: android. app. ContextImpl. sendBroadcast:1433 com. android. server. ConnectivityService. setLteMobileDataEnabled:2117 android. net. IConnectivityManager$Stub. onTransact:987 android. os. Binder. execTransact:446 <bottom of call stack> D/ConnectivityService(969): [setLteMobileDataEnabled()] The intent of set_network_mode is sent now ! D/ConnectivityService(969): [setLteMobileDataEnabled()] The intent of lte_data_completed_action is sent now with disconnecting_4G ! W/ContextImpl(969): Calling a method in the system process without a qualified user: android. app. ContextImpl. sendBroadcast:1433 com. android. server. ConnectivityService. setLteMobileDataEnabled:2128 android. net. IConnectivityManager$Stub. onTransact:987 android. os. Binder. execTransact:446 <bottom of call stack> D/ServiceApiTest::CS(12765): Result = null D/ServiceApiTest::MA(12765): ... finished.	May require certain user to send broadcast?

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ConnectivityService	*setRoamingDataEnabled_RILCMD()*	*None*	Y	`D/ServiceApiTest::MA(26735): Invoking test method... D/ServiceApiTest::SS(26735): Returned object = android. net. IConnectivityManager $Stub $Proxy@4591550 D/ServiceApiTest::CS(26735): Returned object = android. net. IConnectivityManager $Stub $Proxy@4591550 D/ServiceApiTest::CS(26735): Found method = public void android. net. IConnectivityManager $Stub $Proxy. setRoamingDataEnabled_RILCMD(boolean) throws android. os. RemoteException D/ConnectivityService(969): [LG_DATA] setRoamingDataEnabled_RILCMD, enabled = 1 D/ServiceApiTest::CS(26735): Result = null D/ServiceApiTest::MA(26735): ... finished. D/ServiceApiTest::MA(27570): Invoking test method... I/DPLogger(27534): DPLogger - Name was too long. Truncating; original name: BaseMetricsServiceFactory, truncated name: BaseMetricsServiceFacto I/DPLogger(27534): DPLogger - Name was too long. Truncating; original name: MetricsConfigurationParser, truncated name: MetricsConfigurationPar I/MetricsConfigurationPar(27534): MetricsConfigurationParser - Picking configuration; Domain: prod, Build: user, isDebuggable: false D/ServiceApiTest::SS(27570): Returned object = android. net. IConnectivityManager $Stub $Proxy@12dc3a02 D/ServiceApiTest::CS(27570): Returned object = android. net. IConnectivityManager $Stub $Proxy@12dc3a02 D/ServiceApiTest::CS(27570): Found method = public void android. net. IConnectivityManager $Stub $Proxy. setRoamingDataEnabled_RILCMD(boolean) throws android. os. RemoteException D/ConnectivityService(969): [LG_DATA] setRoamingDataEnabled_RILCMD, enabled = 0 D/ServiceApiTest::CS(27570): Result = null D/ServiceApiTest::MA(27570): ... finished.`	Not sure what this did. Need to also look at radio logcat buffer.
LGEncryptionService	*lockDevice()*	*CRYPT_KEEPER*	N		System permission
MHPService	*enable(), disable()*	*None*	Y		Able to turn on/off mobile hotspot without any permissions. Should be protected with CHANGE_NETWORK_STATE and/or ACCESS_NETWORK_STATE.
MHPService	*getName()*	*None*	Y		Returns current AP name. Should be protected with ACCESS_NETWORK_STATE.

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
MHPService	*getWPAKey()*	*None*	Y	`D/ServiceApiTest::MA(23244): Invoking test method... D/ServiceApiTest::SS(23244): Returned object = com. lge. wifi. impl. mobilehotspot. IMobileHotspot $Stub $Proxy@12dc3a02 D/ServiceApiTest::MHP(23244): Returned object = com. lge. wifi. impl. mobilehotspot. IMobileHotspot $Stub $Proxy@12dc3a02 D/ServiceApiTest::MHP(23244): Found method = public java. lang. String com. lge. wifi. impl. mobilehotspot. IMobileHotspot $Stub $Proxy. getWPAKey() throws android. os. RemoteException E/WifiHostapdWrapperBcm(2081): p2pGetWPAKey [000000] D/ServiceApiTest::MHP(23244): Result = 000000 D/ServiceApiTest::MA(23244): ... finished.`	Returns current WPA key for hotspot. Should only be accessible to system uid.
MHPService	*setWPAKey()*	*None*	Y		Should be protected with CHANGE_NETWORK_STATE and/or ACCESS_NETWORK_STATE.
MHPService	*setName*	*None*	Y	`D/ServiceApiTest::MA(17207): Invoking test method... D/ServiceApiTest::SS(17207): Returned object = com. lge. wifi. impl. mobilehotspot. IMobileHotspot $Stub $Proxy@27f1c5f D/ServiceApiTest::MHP(17207): Returned object = com. lge. wifi. impl. mobilehotspot. IMobileHotspot $Stub $Proxy@27f1c5f D/ServiceApiTest::MHP(17207): Found method = public boolean com. lge. wifi. impl. mobilehotspot. IMobileHotspot $Stub $Proxy. setName(java. lang. String) throws android. os. RemoteException D/WifiHostapdWrapperBcm(2081): p2pSetSSID: Pwn3d E/WifiHostapdWrapperBcm(2081): p2pSetSSID : [Pwn3d] D/ServiceApiTest::MHP(17207): Result = true D/ServiceApiTest::MA(17207): ... finished.`	Set AP name (SSID). Should be protected with CHANGE_NETWORK_STATE and/or ACCESS_NETWORK_STATE.

continued...

Method-level test results for LG-5.0.2 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
WiFiOffloadingService	*disableWifi()*	*None*	Y	`D/ServiceApiTest::MA(14233): Invoking test method... D/ServiceApiTest::SS(14233): Returned object = com. lge. wifi. impl. offloading. IWiFiOffloading $Stub $Proxy@1c1df6f0 D/ServiceApiTest::WiFiOLS(14233): Returned object = com. lge. wifi. impl. offloading. IWiFiOffloading $Stub $Proxy@1c1df6f0 D/ServiceApiTest::WiFiOLS(14233): Found method = public void com. lge. wifi. impl. offloading. IWiFiOffloading $Stub $Proxy. disableWifi() throws android. os. RemoteException D/WiFiOffloadingService(3169): [NEZZIMOM] setOffloadOngoing » 0 D/WifiServiceImplEx(1040): setWifiEnabled: false pid=3169, uid=1000, package= android. uid. system:1000, App Lable : null D/WifiServiceImplEx(1040): disconnect pid=3169, uid=1000, packageName=android. uid. system:1000 D/WifiServiceImplEx(1040): checkVZWFeaturehiddenwifi enabled: falsehide_wifi: false D/WifiStateMachine(1040): handleMessage: E msg. what=131145 D/WifiStateMachine(1040): processMsg: ConnectedState E/WifiServiceImplEx(1040): WIFI_AGGREGATION_STOPfalse W/ContextImpl(1040): Calling a method in the system process without a qualified user: android. app. ContextImpl. sendBroadcast:1433 com. android. server. wifi. WifiServiceImplEx. setWifiEnabled:629 android. net. wifi. IWifiManager$Stub. onTransact:222 android. os. Binder. execTransact:446 <bottom of call stack> E/WifiStateMachine(1040): ConnectedState !CMD_DISCONNECT 0 0 D/WifiStateMachine(1040): processMsg: L2ConnectedState D/WifiService(1040): setWifiEnabled: false pid=3169, uid=1000 E/WifiStateMachine(1040): L2ConnectedState !CMD_DISCONNECT 0 0 E/WifiNative: (1040): [4,027,256,084 us] DISCONNECT stack:logDbg - disconnect - processMessage - processMsg - handleMessage D/WifiNative-wlan0(1040): doBoolean: DISCONNECT D/wpa_supplicant(4210): wlan0: Control interface command 'DISCONNECT' D/wpa_supplicant(4210): wlan0: Cancelling scan request D/wpa_supplicant(4210): wlan0: Request to deauthenticate - bssid=f0:9c:e9:13:69:16 pending_bssid=00:00:00:00:00:00 reason=3 state=COMPLETED D/wpa_supplicant(4210): TDLS: Tear down peers D/wpa_supplicant(4210): wpa_driver_nl80211_disconnect(reason_code=3) D/ServiceApiTest::WiFiOLS(14233): Result = null D/ServiceApiTest::MA(14233): ... finished.`	Other unprotected methods that should be protected with CHANGE_NETWORK_STATE and/or ACCESS_NETWORK_STATE.

Appendix H

S4-5.0.1 Test Results

Table H.1: Method-level test results for S4-5.0.1.

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ABTPersistanceService	*getDeviceId()*	*None*	N	D/ServiceApiTest::MA(2286): Invoking test method... D/ServiceApiTest::SS(2286): Returned object = com. absolute. android. persistence. IABTPersistence $Stub $Proxy@274b8d19 D/ServiceApiTest::ABTPS(2286): Returned object = com. absolute. android. persistence. IABTPersistence $Stub $Proxy@274b8d19 D/ServiceApiTest::ABTPS(2286): Found method = public java. lang. String com. absolute. android. persistence. IABTPersistence $Stub $Proxy. getDeviceId() throws android. os. RemoteException E/ServiceApiTest::MA(2286): java. lang. reflect. InvocationTargetException W/System. err(2286): java. lang. reflect. InvocationTargetException W/System. err(2286): at java. lang. reflect. Method. invoke(Native Method) W/System. err(2286): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(2286): at com. ratazzi. serviceapitest. ABTPersistenceService. testMethodCall(ABTPersistenceService. java:43) W/System. err(2286): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:25) W/System. err(2286): at android. app. Activity. performCreate(Activity. java:6289) W/System. err(2286): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1119) W/System. err(2286): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2646) W/System. err(2286): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2758) W/System. err(2286): at android. app. ActivityThread. access$900(ActivityThread. java:177) W/System. err(2286): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1448) W/System. err(2286): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(2286): at android. os. Looper. loop(Looper. java:145) W/System. err(2286): at android. app. ActivityThread. main(ActivityThread. java:5942) W/System. err(2286): at java. lang. reflect. Method. invoke(Native Method) W/System. err(2286): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(2286): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:1400) W/System. err(2286): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:1195) W/System. err(2286): Caused by: java. lang. SecurityException: Not authorized to access ABT Persistence Service W/System. err(2286): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(2286): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(2286): at com. absolute. android. persistence. IABTPersistence $Stub $Proxy. getDeviceId(IABTPersistence. java:630) W/System. err(2286): ... 17 more	Private method g() enforces uid==1000. Other AIDL methods protected in same way, so not tested.

continued...

Method-level test results for S4-5.0.1 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
AccessibilityManagerService	*enableMagnifier()*	*None*	Y	D/ServiceApiTest::MA(12334): Invoking test method... D/ServiceApiTest::SS(12334): Returned object = android. view. accessibility. IAccessibilityManager $Stub $Proxy@3b2a44cf D/ServiceApiTest::A11yMS(12334): Returned object = android. view. accessibility. IAccessibilityManager $Stub $Proxy@3b2a44cf D/ServiceApiTest::A11yMS(12334): Found method = public void android. view. accessibility. IAccessibilityManager $Stub $Proxy. enableMagnifier(int,int,float) throws android. os. RemoteException D/DisplayManagerService(1011): enableOverlayMagnifier:true D/LensFlare(1621): LensFlare sound : release V/AudioPolicyManager_legacy(306): releaseOutput() 2 D/MagnifierDisplayAdapter(1011): enable:true V/AudioPolicyManager_legacy(306): releaseOutput() 2 D/SurfaceFlinger(281): FPS : 60. 52 W/SurfaceFlinger(281): Fail to Open /sys/devices/platform/gpusysfs/fps D/MagnifierDisplayAdapter(1011): onCreateSurface:Surface(name=null)/@0x397a5f76 D/MagnifierDevice(1011): MagnifierDevice init D/DisplayManagerService(1011): setMagnificationSettings scale:10. 0 (10, 10) I/DisplayManagerService(1011): Display device added: DisplayDeviceInfo‘‘OverlayMagnifier”: 1080 x 1920, 60. 0 fps, supportedRefreshRates [], density 480, 480. 0 x 480. 0 dpi, appVsyncOff 0, presDeadline 0, touch NONE, rotation 0, type OVERLAY, state ON, FLAG_PRESENTATION V/ActivityManager(1011): Display added displayId=1 D/CoverManager(1011): onCoverAppCovered!! D/ServiceApiTest::A11yMS(12334): Result = null D/ServiceApiTest::MA(12334): ... finished.	
AccessibilityManagerService	*disableMagnifier()*	*None*	Y	D/ServiceApiTest::MA(16008): Invoking test method... W/ActivityManager(1011): Activity pause timeout for ActivityRecord1646175d u0 com. ratazzi. serviceapitest/. MainActivity t27 D/SurfaceFlinger(281): FPS : 62. 33 W/SurfaceFlinger(281): Fail to Open /sys/devices/platform/gpusysfs/fps D/ServiceApiTest::SS(16008): Returned object = android. view. accessibility. IAccessibilityManager $Stub $Proxy@3b2a44cf D/ServiceApiTest::A11yMS(16008): Returned object = android. view. accessibility. IAccessibilityManager $Stub $Proxy@3b2a44cf D/ServiceApiTest::A11yMS(16008): Found method = public void android. view. accessibility. IAccessibilityManager $Stub $Proxy. disableMagnifier() throws android. os. RemoteException D/DisplayManagerService(1011): enableOverlayMagnifier:false D/MagnifierDisplayAdapter(1011): enable:false I/DisplayManagerService(1011): chipname info: MSM8960 I/DisplayManagerService(1011): Display device removed: DisplayDeviceInfo‘‘OverlayMagnifier”: 1080 x 1920, 60. 0 fps, supportedRefreshRates [], density 480, 480. 0 x 480. 0 dpi, appVsyncOff 0, presDeadline 0, touch NONE, rotation 0, type OVERLAY, state ON, FLAG_PRESENTATION V/ActivityManager(1011): Display removed displayId=1 D/ServiceApiTest::A11yMS(16008): Result = null D/CoverManager(1011): onCoverAppCovered!! D/SurfaceFlinger(281): FPS : 62. 33 W/SurfaceFlinger(281): Fail to Open /sys/devices/platform/gpusysfs/fps D/ServiceApiTest::MA(16008): ... finished.	

continued...

Method-level test results for S4-5.0.1 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
AccessibilityManagerService	*reboot()*	*REBOOT*	Y	D/ServiceApiTest::MA(18611): Invoking test method... D/ServiceApiTest::SS(18611): Returned object = android. view. accessibility. IAccessibilityManager $Stub $Proxy@19108e5c D/ServiceApiTest::A11yMS(18611): Returned object = android. view. accessibility. IAccessibilityManager $Stub $Proxy@19108e5c D/ServiceApiTest::A11yMS(18611): Found method = public void android. view. accessibility. IAccessibilityManager $Stub $Proxy. reboot(boolean) throws android. os. RemoteException D/PowerManagerService(1011): !@ reason: Assistant Menu confirm: false wait: false (uid: 10195 pid: 18611 processName: com. ratazzi. serviceapitest) E/ServiceApiTest::MA(18611): java. lang. reflect. InvocationTargetException W/System. err(18611): java. lang. reflect. InvocationTargetException W/System. err(18611): at java. lang. reflect. Method. invoke(Native Method) W/System. err(18611): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(18611): at com. ratazzi. serviceapitest. AccessibilityManagerService. testMethodCall(AccessibilityManagerService. java:47) W/System. err(18611): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:25) W/System. err(18611): at android. app. Activity. performCreate(Activity. java:6289) W/System. err(18611): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1119) W/System. err(18611): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2646) W/System. err(18611): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2758) W/System. err(18611): at android. app. ActivityThread. access$900(ActivityThread. java:177) W/System. err(18611): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1448) W/System. err(18611): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(18611): at android. os. Looper. loop(Looper. java:145) W/System. err(18611): at android. app. ActivityThread. main(ActivityThread. java:5942) W/System. err(18611): at java. lang. reflect. Method. invoke(Native Method) W/System. err(18611): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(18611): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:1400) W/System. err(18611): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:1195) W/ActivityManager(1011): Activity pause timeout for ActivityRecord380878f9 u0 com. ratazzi. serviceapitest/. MainActivity t29 W/System. err(18611): Caused by: java. lang. SecurityException: Neither user 10195 nor current process has android. permission. REBOOT. W/System. err(18611): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(18611): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(18611): at android. view. accessibility. IAccessibilityManager $Stub $Proxy. reboot(IAccessibilityManager. java:746) W/System. err(18611): ... 17 more	System permission.

continued...

Method-level test results for S4-5.0.1 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
WindowManagerService	*lockNow()*	*DEVICE_POWER*	Y	`D/ServiceApiTest::MA(29606): Invoking test method... D/ServiceApiTest::SS(29606): Returned object = android. view. IWindowManager $Stub $Proxy@274b8d19 D/ServiceApiTest::WMS(29606): Returned object = android. view. IWindowManager $Stub $Proxy@274b8d19 D/ServiceApiTest::WMS(29606): Found method = public void android. view. IWindowManager $Stub $Proxy. lockNow(android. os. Bundle) throws android. os. RemoteException E/ServiceApiTest::MA(29606): java. lang. reflect. InvocationTargetException W/System. err(29606): java. lang. reflect. InvocationTargetException W/System. err(29606): at java. lang. reflect. Method. invoke(Native Method) W/System. err(29606): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(29606): at com. ratazzi. serviceapitest. WindowManagerService. testMethodCall(WindowManagerService. java:48) W/System. err(29606): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:25) W/System. err(29606): at android. app. Activity. performCreate(Activity. java:6289) W/System. err(29606): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1119) W/System. err(29606): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2646) W/System. err(29606): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2758) W/System. err(29606): at android. app. ActivityThread. access$900(ActivityThread. java:177) W/System. err(29606): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1448) W/System. err(29606): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(29606): at android. os. Looper. loop(Looper. java:145) W/System. err(29606): at android. app. ActivityThread. main(ActivityThread. java:5942) W/System. err(29606): at java. lang. reflect. Method. invoke(Native Method) W/System. err(29606): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(29606): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:1400) W/System. err(29606): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:1195) W/System. err(29606): Caused by: java. lang. SecurityException: Neither user 10195 nor current process has android. permission. DEVICE_POWER. W/System. err(29606): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(29606): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(29606): at android. view. IWindowManager $Stub $Proxy. lockNow(IWindowManager. java:2521) W/System. err(29606): ... 17 more`	System permission.

continued...

Method-level test results for S4-5.0.1 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
AudioService	*adjustMasterVolume()*	*None*	Y	`D/ServiceApiTest::MA(16520): Invoking test method... D/ServiceApiTest::SS(16520): Returned object = android. media. IAudioService $Stub $Proxy@3a7ee1de D/ServiceApiTest::AS(16520): Returned object = android. media. IAudioService $Stub $Proxy@3a7ee1de D/ServiceApiTest::AS(16520): Found method = public void android. media. IAudioService $Stub $Proxy. setMasterVolume(int,int,java. lang. String) throws android. os. RemoteException D/VolumePanel(1621): onVolumeChanged : call resetTimeout() D/VolumePanel(1621): resetTimeout : call sendEmptyMessageDelayed : MSG_TIMEOUT D/PowerManagerService(1011): [api] [s] userActivity : event: 0 flags: 0 (uid: 10068 pid: 1621) eventTime = 20521022 D/ServiceApiTest::AS(16520): Result = null D/ServiceApiTest::MA(16520): ... finished.`	Effect not evident.
AudioService	*playSoundEffect()*	*None*	Y	`D/ServiceApiTest::MA(22609): Invoking test method... D/ServiceApiTest::SS(22609): Returned object = android. media. IAudioService $Stub $Proxy@3b2a44cf D/ServiceApiTest::AS(22609): Returned object = android. media. IAudioService $Stub $Proxy@3b2a44cf D/ServiceApiTest::AS(22609): Found method = public void android. media. IAudioService $Stub $Proxy. playSoundEffect(int) throws android. os. RemoteException D/ServiceApiTest::AS(22609): Result = null D/ServiceApiTest::MA(22609): ... finished.`	Seems to adjust sound effect volume for launching activities
AudioService	*setMasterVolume()*	*None*	Y	`D/ServiceApiTest::MA(3770): Invoking test method... D/SurfaceFlinger(281): FPS : 59. 71 D/ServiceApiTest::SS(3770): Returned object = android. media. IAudioService $Stub $Proxy@3b2a44cf D/ServiceApiTest::AS(3770): Returned object = android. media. IAudioService $Stub $Proxy@3b2a44cf D/ServiceApiTest::AS(3770): Found method = public void android. media. IAudioService $Stub $Proxy. setMasterVolume(int,int,java. lang. String) throws android. os. RemoteException W/SurfaceFlinger(281): Fail to Open /sys/devices/platform/gpusysfs/fps D/VolumePanel(1621): onVolumeChanged : call resetTimeout() D/VolumePanel(1621): resetTimeout : call sendEmptyMessageDelayed : MSG_TIMEOUT D/PowerManagerService(1011): [api] [s] userActivity : event: 0 flags: 0 (uid: 10068 pid: 1621) eventTime = 21490816 D/ServiceApiTest::AS(3770): Result = null D/ServiceApiTest::MA(3770): ... finished.`	
ClipboardService	*getClipedStrings()*	*None*	Y	`D/ServiceApiTest::MA(11821): Invoking test method... D/CoverManager(1011): onCoverAppCovered!! D/SurfaceFlinger(281): FPS : 41. 36 W/SurfaceFlinger(281): Fail to Open /sys/devices/platform/gpusysfs/fps D/ServiceApiTest::SS(11821): Returned object = android. sec. clipboard. IClipboardService $Stub $Proxy@3b2a44cf D/ServiceApiTest::CBExS(11821): Returned object = android. sec. clipboard. IClipboardService $Stub $Proxy@3b2a44cf D/ServiceApiTest::CBExS(11821): Found method = public java. util. ArrayList android. sec. clipboard. IClipboardService $Stub $Proxy. getClipedStrings(int,int) throws android. os. RemoteException I/ClipboardServiceEx(1011): getPersonaId api current userid : 0 I/ClipboardServiceEx(1011): isKnoxTwoEnabled getPersonaId 0 I/ClipboardServiceEx(1011): Current user is a USER, hence return false D/ServiceApiTest::CBExS(11821): Result = [] D/ServiceApiTest::MA(11821): ... finished.`	

continued...

Method-level test results for S4-5.0.1 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
ClipboardService	*getPrimaryClip()*	*None*	Y	`D/ServiceApiTest::MA(10501): Invoking test method... D/ServiceApiTest::SS(10501): Returned object = android. sec. clipboard. IClipboardService $Stub $Proxy@274b8d19 D/ServiceApiTest::CBExS(10501): Returned object = android. sec. clipboard. IClipboardService $Stub $Proxy@274b8d19 D/ServiceApiTest::CBExS(10501): Found method = public java. util. ArrayList android. sec. clipboard. IClipboardService $Stub $Proxy. getClipedStrings(int,int) throws android. os. RemoteException I/ClipboardServiceEx(1011): getPersonaId api current userid : 0 I/ClipboardServiceEx(1011): isKnoxTwoEnabled getPersonaId 0 I/ClipboardServiceEx(1011): Current user is a USER, hence return false D/ServiceApiTest::CBExS(10501): Result = H4sIAAAAAAAAAA== D/ServiceApiTest::MA(10501): ... finished.`	

Appendix I

MotoX-5.0 Test Results

Table I.1: Method-level test results for MotoX-5.0.

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*addUpstreamV6Interface()*	*ACCESS_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(4739): Invoking test method... D/ServiceApiTest::SS(4739): Returned object = android. os. INetworkManagementService $Stub $Proxy@192d3c01 D/ServiceApiTest::NMS(4739): Returned object = android. os. INetworkManagementService $Stub $Proxy@192d3c01 D/ServiceApiTest::NMS(4739): Found method = public void android. os. INetworkManagementService $Stub $Proxy. addUpstreamV6Interface(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(922): addUpstreamInterface(abc0) D/CommandListener(358): TetherCmd::runCommand. argc: 4. argv[0]: tether D/CommandListener(358): command tether interface add_upstream abc0 D/TetherController(358): addUpstreamInterface(abc0) D/TetherController(358): addV6RtrAdvIface: len = 1. Iface: abc0 D/TetherController(358): Router advertisement daemon already stopped D/TetherController(358): Router advertisement daemon running D/ServiceApiTest::MA(4739): ...finished.`	

continued…

Method-level test results for MotoX-5.0 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*blockDataTrafficInternal()*	*None*	N	D/ServiceApiTest::MA(5393): Invoking test method... I/ConfigService(1939): onBind returning update interface I/ConfigService(1939): onBind for Intent act=com. google. android. gms. config. START pkg=com. google. android. gms action com. google. android. gms. config. START I/ConfigService(1939): onBind returning config service I/ConfigFetchService(1910): onStartCommand Intent act=android. intent. action. PACKAGE_ADDED dat=package:com. ratazzi. serviceapitest cmp=com. google. android. gms/. config. ConfigFetchService (has extras) I/ConfigFetchService(1910): launchTask D/ServiceApiTest::SS(5393): Returned object = android. os. INetworkManagementService $Stub $Proxy@268a3266 D/ServiceApiTest::NMS(5393): Returned object = android. os. INetworkManagementService $Stub $Proxy@268a3266 D/ServiceApiTest::NMS(5393): Found method = public void android. os. INetworkManagementService $Stub $Proxy. blockDataTrafficInternal(boolean) throws android. os. RemoteException I/ConfigFetchService(1910): service connected I/ConfigClient(1910): service connected E/ServiceApiTest::MA(5393): java. lang. reflect. InvocationTargetException W/System. err(5393): java. lang. reflect. InvocationTargetException W/System. err(5393): at java. lang. reflect. Method. invoke(Native Method) W/System. err(5393): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(5393): at com. ratazzi. serviceapitest. NetworkManagementService. testMethodCall(NetworkManagementService. java:47) W/System. err(5393): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:26) W/System. err(5393): at android. app. Activity. performCreate(Activity. java:5953) W/System. err(5393): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1128) W/System. err(5393): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2267) W/System. err(5393): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2388) W/System. err(5393): at android. app. ActivityThread. access$800(ActivityThread. java:148) W/System. err(5393): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1292) W/System. err(5393): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(5393): at android. os. Looper. loop(Looper. java:135) W/System. err(5393): at android. app. ActivityThread. main(ActivityThread. java:5312) W/System. err(5393): at java. lang. reflect. Method. invoke(Native Method) W/System. err(5393): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(5393): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:901) W/System. err(5393): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:696) W/System. err(5393): Caused by: java. lang. SecurityException: Only available to AID_SYSTEM W/System. err(5393): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(5393): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(5393): at android. os. INetworkManagementService $Stub $Proxy. blockDataTrafficInternal(INetworkManagementService. java:2660) W/System. err(5393): ... 17 more	Requires uid=1000 (AID_SYSTEM)

continued...

Method-level test results for MotoX-5.0 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*enableTrafficMonitor()*				System permission
NetworkManagementService	*getSapAutoChannelSelection()*	*CHANGE_NETWORK_STATE* *CHANGE_WIFI_STATE*	Y	`D/ServiceApiTest::MA(20119): Invoking test method... D/ServiceApiTest::SS(20119): Returned object = android. os. INetworkManagementService $Stub $Proxy@11b3afa8 D/ServiceApiTest::NMS(20119): Returned object = android. os. INetworkManagementService $Stub $Proxy@11b3afa8 D/ServiceApiTest::NMS(20119): Found method = public int android. os. INetworkManagementService $Stub $Proxy. getSapAutoChannelSelection() throws android. os. RemoteException D/NetworkManagementService(933): getSapAutoChannelSelection D/ (343): CMD INPUT [get autochannel][256] E/ (343): qsap_read_auto_channel : ... D/QCLDR- (343): HOSTAPD disabled W/ps (20150): type=1400 audit(0. 0:426): avc: denied search for uid=0 name=“12” dev=“proc” ino=8444 scontext=u:r:netd:s0 tcontext=u:r:kernel:s0 tclass=dir permissive=0 E/ (343): qsap_get_sap_auto_channel_selection :is_softap_enabled() goto error E/ (343): qsap_get_sap_auto_channel_selection: Failed to read sap auto channel selection D/ (343): CMD OUTPUT [failure unknown error] D/ (343): len :21 D/ (343): D/NetworkManagementService(933): getSapAutoChannelSelection-OperChanResp200 99 failure unknown error D/NetworkManagementService(933): softap qccmd get autochannel =0 D/ServiceApiTest::NMS(20119): Result = 0 D/ServiceApiTest::MA(20119): ...finished.`	Seems to have SELinux denials
NetworkManagementService	*getSapOperatingChannel()*	*CHANGE_NETWORK_STATE* *CHANGE_WIFI_STATE*	Y	`D/ServiceApiTest::MA(23700): Invoking test method... D/ServiceApiTest::SS(23700): Returned object = android. os. INetworkManagementService $Stub $Proxy@11b3afa8 D/ServiceApiTest::NMS(23700): Returned object = android. os. INetworkManagementService $Stub $Proxy@11b3afa8 D/ServiceApiTest::NMS(23700): Found method = public int android. os. INetworkManagementService $Stub $Proxy. getSapOperatingChannel() throws android. os. RemoteException D/NetworkManagementService(933): getSapOperatingChannel D/ (343): CMD INPUT [get channel][256] ... D/QCLDR- (343): HOSTAPD disabled W/ps (23736): type=1400 audit(0. 0:799): avc: denied search for uid=0 name=“8” dev=“proc” ino=8440 scontext=u:r:netd:s0 tcontext=u:r:kernel:s0 tclass=dir permissive=0 E/ (343): qsap_get_operating_channel: Failed to read channel D/ (343): CMD OUTPUT [failure unknown error] D/ (343): len :21 D/ (343): D/NetworkManagementService(933): getSapOperatingChannel-OperChanResp200 100 failure unknown error D/NetworkManagementService(933): softap qccmd get channel =0 D/ServiceApiTest::NMS(23700): Result = 0 D/ServiceApiTest::MA(23700): ...finished.`	Seems to have SELinux denials

continued...

Method-level test results for MotoX-5.0 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*removeUpstreamV6Interface()*	*ACCESS_NETWORK_STATE*	Y	`D/ServiceApiTest::MA(28640): Invoking test method... D/ServiceApiTest::SS(28640): Returned object = android. os. INetworkManagementService $Stub $Proxy@268a3266 D/ServiceApiTest::NMS(28640): Returned object = android. os. INetworkManagementService $Stub $Proxy@268a3266 D/ServiceApiTest::NMS(28640): Found method = public void android. os. INetworkManagementService $Stub $Proxy. removeUpstreamV6Interface(java. lang. String) throws android. os. RemoteException D/NetworkManagementService(933): removeUpstreamInterface(abc0) D/CommandListener(343): TetherCmd::runCommand. argc: 4. argv[0]: tether W/TetherController(343): Couldn’t find interface abc0 to remove D/ServiceApiTest::NMS(28640): Result = null D/ServiceApiTest::MA(28640): ...finished.`	

continued...

Method-level test results for MotoX-5.0 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*runIpLogCmd()*	*None*	Y	`D/ServiceApiTest::MA(17711): Invoking test method... D/ServiceApiTest::SS(17711): Returned object = android. os. INetworkManagementService $Stub $Proxy@11b3afa8 D/ServiceApiTest::NMS(17711): Returned object = android. os. INetworkManagementService $Stub $Proxy@11b3afa8 D/ServiceApiTest::NMS(17711): Found method = public int android. os. INetworkManagementService $Stub $Proxy. runIpLogCmd(java. lang. String) throws android. os. RemoteException D/IpLogController(343): runRawCmd E/IpLogController(343): Unsupported iplog cmd: (null) E/ServiceApiTest::MA(17711): java. lang. reflect. InvocationTargetException W/System. err(17711): java. lang. reflect. InvocationTargetException W/System. err(17711): at java. lang. reflect. Method. invoke(Native Method) W/System. err(17711): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(17711): at com. ratazzi. serviceapitest. NetworkManagementService. testMethodCall(NetworkManagementService. java:52) W/System. err(17711): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:26) W/System. err(17711): at android. app. Activity. performCreate(Activity. java:5953) W/System. err(17711): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1128) W/System. err(17711): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2267) W/System. err(17711): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2388) W/System. err(17711): at android. app. ActivityThread. access$800(ActivityThread. java:148) W/System. err(17711): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1292) W/System. err(17711): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(17711): at android. os. Looper. loop(Looper. java:135) W/System. err(17711): at android. app. ActivityThread. main(ActivityThread. java:5312) W/System. err(17711): at java. lang. reflect. Method. invoke(Native Method) W/System. err(17711): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(17711): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:901) W/System. err(17711): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:696) W/System. err(17711): Caused by: java. lang. IllegalStateException: command ’161 iplog capture’ failed with ’400 161 IpLog operation failed (Success)’ W/System. err(17711): at android. os. Parcel. readException(Parcel. java:1548) W/System. err(17711): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(17711): at android. os. INetworkManagementService $Stub $Proxy. runIpLogCmd(INetworkManagementService. java:2277) W/System. err(17711): ... 17 more`	IpLogController reports execution but don’t know what commands are supported. Possibility is there to capture network traffic.

continued...

Method-level test results for MotoX-5.0 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
NetworkManagementService	*setChannelRange()*	*CHANGE_NETWORK_STATE* *CHANGE_WIFI_STATE*	Y	`D/ServiceApiTest::MA(25369): Invoking test method... D/ServiceApiTest::SS(25369): Returned object = android. os. INetworkManagementService $Stub $Proxy@268a3266 D/ServiceApiTest::NMS(25369): Returned object = android. os. INetworkManagementService $Stub $Proxy@268a3266 D/ServiceApiTest::NMS(25369): Found method = public void android. os. INetworkManagementService $Stub $Proxy. setChannelRange(int,int,int) throws android. os. RemoteException D/NetworkManagementService(933): Set SAP Channel Range D/ (343): CMD INPUT [set setchannelrange= 2 3 5][256] E/ (343): Cmd: setchannelrange Argument : 2 3 5 D/ (343): cmd=setchannelrange, Val: 2 3 5, INI:0 D/ (343): Updated:setchannelrange= 2 3 5 D/ (343): D/ (343): CMD OUTPUT [success] D/ (343): len :7 D/ (343): D/ServiceApiTest::NMS(25369): Result = null D/ServiceApiTest::MA(25369): ...finished.`	
PowerManagerService	*proximityTargetDetected()*	*None*	Y	`D/ServiceApiTest::MA(31671): Invoking test method... D/ServiceApiTest::SS(31671): Returned object = android. os. IPowerManager $Stub $Proxy@11b3afa8 D/ServiceApiTest::PMS(31671): Returned object = android. os. IPowerManager $Stub $Proxy@11b3afa8 D/ServiceApiTest::PMS(31671): Found method = public boolean android. os. IPowerManager $Stub $Proxy. proximityTargetDetected() throws android. os. RemoteException D/ServiceApiTest::PMS(31671): Result = false D/ServiceApiTest::MA(31671): ...finished.`	

continued...

Method-level test results for MotoX-5.0 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
PowerManagerService	*setQuickOff()*	*com. motorola. permission. EXTRA_SCREEN_CONTROL*	N	`D/ServiceApiTest::MA(7461): Invoking test method... D/ServiceApiTest::SS(7461): Returned object = android. os. IPowerManager $Stub $Proxy@268a3266 D/ServiceApiTest::PMS(7461): Returned object = android. os. IPowerManager $Stub $Proxy@268a3266 D/ServiceApiTest::PMS(7461): Found method = public void android. os. IPowerManager $Stub $Proxy. setQuickOff(float) throws android. os. RemoteException E/ServiceApiTest::MA(7461): java. lang. reflect. InvocationTargetException W/System. err(7461): java. lang. reflect. InvocationTargetException W/System. err(7461): at java. lang. reflect. Method. invoke(Native Method) W/System. err(7461): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(7461): at com. ratazzi. serviceapitest. PowerManagerService. testMethodCall(PowerManagerService. java:48) W/System. err(7461): at com. ratazzi. serviceapitest. MainActivity. onCreate(MainActivity. java:26) W/System. err(7461): at android. app. Activity. performCreate(Activity. java:5953) W/System. err(7461): at android. app. Instrumentation. callActivityOnCreate(Instrumentation. java:1128) W/System. err(7461): at android. app. ActivityThread. performLaunchActivity(ActivityThread. java:2267) W/System. err(7461): at android. app. ActivityThread. handleLaunchActivity(ActivityThread. java:2388) W/System. err(7461): at android. app. ActivityThread. access$800(ActivityThread. java:148) W/System. err(7461): at android. app. ActivityThread$H. handleMessage(ActivityThread. java:1292) W/System. err(7461): at android. os. Handler. dispatchMessage(Handler. java:102) W/System. err(7461): at android. os. Looper. loop(Looper. java:135) W/System. err(7461): at android. app. ActivityThread. main(ActivityThread. java:5312) W/System. err(7461): at java. lang. reflect. Method. invoke(Native Method) W/System. err(7461): at java. lang. reflect. Method. invoke(Method. java:372) W/System. err(7461): at com. android. internal. os. ZygoteInit$MethodAndArgsCaller. run(ZygoteInit. java:901) W/System. err(7461): at com. android. internal. os. ZygoteInit. main(ZygoteInit. java:696) W/System. err(7461): Caused by: java. lang. SecurityException: Neither user 10115 nor current process has com. motorola. permission. EXTRA_SCREEN_CONTROL. W/System. err(7461): at android. os. Parcel. readException(Parcel. java:1540) W/System. err(7461): at android. os. Parcel. readException(Parcel. java:1493) W/System. err(7461): at android. os. IPowerManager $Stub $Proxy. setQuickOff(IPowerManager. java:781) W/System. err(7461): ... 17 more`	System permission: W/PackageManager(933): Not granting permission com. motorola. permission. EXTRA_SCREEN_CONTROL to package com. ratazzi. serviceapitest (protectionLevel=18 flags=0x48be46)

continued...

Method-level test results for MotoX-5.0 (continued).

Service Name	Method Name	Permission(s) Needed	Invoke?	Logcat	Notes
PowerManagerService	*setPreDim()*	*com. motorola. permission. EXTRA_SCREEN_CONTROL*	N		System permission: W/PackageManager(933): Not granting permission com. motorola. permission. EXTRA_SCREEN_CONTROL to package com. ratazzi. serviceapitest (protectionLevel=18 flags=0x48be46)
PowerManagerService	*setQuickDim()*	*com. motorola. permission. EXTRA_SCREEN_CONTROL*	N		System permission: W/PackageManager(933): Not granting permission com. motorola. permission. EXTRA_SCREEN_CONTROL to package com. ratazzi. serviceapitest (protectionLevel=18 flags=0x48be46)
StatusBarManagerService	*showHomeSearchAction()*	*None*	Y		
StatusBarManagerService	*hideHomeSearchAction()*	*None*	Y		
VzwConnectivityService					Don't know service registration name; service may not exist on this phone since seems to be an AT&T phone.
WindowManagerService	*pauseRotation()*	*None*	N		Requires uid==1000
WindowManagerService	*resumeRotation()*	*None*	N		Requires uid==1000

Appendix J

Linux Namespaces Analysis

This appendix provides a background on Linux Namespaces and the analysis of the traits which offer value to Android security. These traits and values are used to support the design tradeoffs discussed in Chapter 5 and are summarized in Table 5.2.

J.1 Background

Since their introduction to the mainstream Linux kernel beginning in 2002, Linux Namespaces have enabled a form of kernel-enforced process-level virtualization, useful for isolating specific resources within a single instance of the operating system. In contrast to hypervisor-based approaches, Namespaces represent a lightweight, "only what's needed," approach to virtualizing various aspects of a process's environment. This container architecture is arguably more scalable and efficient than full virtualization solutions [71] [70]. Practical uses for Namespaces include resource sharing [67], checkpointing/migration, [68] vulnerability containment, and binary isolation [126].

Simply put, namespaces allow different objects to have the same name. For example, the objects "server1" and "server2" could both have the name "hostname" in different namespaces. In Linux, the original motivation for including namespace support were the needs of virtual private servers

(VPS) and application checkpoint and restart (ACR) [68]. Six namespaces make up the current Linux implementation discussed here: *mount*, *hostinfo*, *System V IPC*, *pid*, *network*, and *userid*. Each of these orthogonal namespaces address a different aspect of the system which can be virtualized for particular applications. They can be used in any combination or individually. Finally, while the main feature of namespaces can be described as lightweight virtualization, namespaces also represent a form of isolation since access to namespaces is controlled by the Linux kernel. Creation and management of Linux namespaces done through privileged system calls.

Namespaces are created and managed by way of three privileged system calls: **`clone()`**, **`unshare()`** and **`setns()`**. **`clone()`** forks a new process into a new namespace, while **`unshare()`** and **`setns()`** allow the calling process to leave and join other namespaces, respectively. Configuration of each new namespace is specified using some twenty-two different **`CLONE_*`** flags to identify the type, and how the caller's execution context is to be shared.

Implementation of namespaces within the Linux kernel is accomplished by way of a proxy structure of pointers, **`nsproxy`**. As namespace support was added to the kernel, system calls that interact with namespace objects were rewritten to use this proxy rather than referencing objects directly. Hence, one process's **`nsproxy`** may point to different objects as compared with another's. From these two processes' point of view, they are using the exact same name to reference the object, but are given back different objects. The object returned is determined by the state of **`nsproxy`**, which is out of the process's control. In a sense, each process is isolated from other objects with the same name in different namespaces. This isolation is immutable because the redirection is enforced by the kernel, within the trusted computing base (TCB).

To illustrate this implementation, we use the *hostinfo*, or *UTS*[1] namespace as an example. UTS namespaces isolate two system identifiers returned by the **`uname()`** system call, namely **`nodename`** and **`domainname`**. Prior to kernel version 2.6.19, these values were returned by reading

[1] This name derives from Unix Timesharing System, abbreviated **`uts`** in the source code of many Unix-like operating systems.

them from global kernel parameters, `system_utsname.nodename` and `system_utsname.domainname`, respectively. As a result, these objects had the same values for all processes. Beginning with the introduction of UTS namespace support in kernel version 2.6.19, these values are read from a private kernel data structure, `nsproxy`. One member of `nsproxy` is a pointer to a structure containing UTS namespace objects such as `nodename` and `domainname`. Processes that have all the same namespaces share `nsproxy`, but it is copied and its pointers changed when one of the namespaces is cloned or unshared. In this way, the `nsproxy` structure enables the kernel to hold different objects (i.e., values) referenced with the same name (e.g., `nodename`, `domainname`) by different processes.

In userspace, namespace support adds a set of symbolic links in `/proc/<pid>/ns`, one for each namespace supported. These links act as “handles” for processes to use to interact with the namespaces. For example, the handles can be compared across two processes to determine if they belong to the same namespace. Also, by passing a file descriptor for one of these links to `setns()`, a privileged process may join the namespace [127].

J.2 Namespace Traits and Their Value to Our Work

With a modified Linux kernel at its heart, and resource constraints that may preclude full virtualization, Android is a great candidate to take advantage of the benefits of Linux Namespaces. In fact, recent versions supporting multiple users and restricted profiles leverage mount namespaces to isolate each user’s external storage [35]. It is likely that future Android releases will make further use of these facilities.

As mentioned earlier, our objective is not to isolate Framework objects using Linux Namespaces themselves, but to develop a specific virtualization and isolation architecture that reflects the same high-level benefits as Linux Namespaces. To do this, we conducted a systematic analysis to extract a list of key traits and link these to specific benefits that our architecture should realize.

The following analysis is summarized in Table 5.2.

Trait 1: *Namespaces virtualize and isolate specific resources on a per-process basis*. Namespaces provide a level of abstraction that wraps certain resources, allowing them to appear to be dedicated to a particular process rather than globally shared [127][128]. These virtual resources are isolated because the kernel controls which processes can join or leave a namespace.

Value to our work: The fine granularity of namespace isolation allows each process to be placed in a tailored virtual environment. Processes spawned by untrusted applications can be presented with a surrogate or subset of sensitive, otherwise global resource(s), thus making it impossible for the application to misuse the resource(s). This can be done permanently for some applications or temporarily while trust is established during testing and debugging [129].

Trait 2: *Namespaces are resource-centric*. When there is a specific resource that must be isolated, namespaces are able to address the isolation directly, unlike coarse-grained approaches such as platform virtualization.

Value to our work: Mobile device users are concerned with their privacy and personal data, and typically think in terms of these high-level semantics rather than trying to understand the details of why an application needs certain permissions and how granting that permission could negatively affect them. At the same time, users want their applications to work smoothly and provide the functionality they desire (but no more). By isolating resources using semantics similar to the user's point of view, namespaces relieve the user from having to understand low-level access control mechanisms and the permission-to-resource mappings. This translates to a high level of usability, not just for the user's configuration of namespaces, but also for the platform as a whole. Moreover, because Android is based on an open architecture that facilitates and depends on a high level of interaction among apps, isolating only certain critical resources preserves other functionality without having to implement complex workarounds or inconveniencing the user.

Trait 3: *Namespaces are efficient*. Linux Namespaces share a single kernel and operating system instance [71][129]. As a result, isolation of a specific resource can be achieved with high efficiency, and overall performance levels are nearly identical those without any isolation [71][70][68][67]. Unlike isolation based on platform or application virtualization, there is no extra startup time for a namespace [71].

Value to our work: By applying the namespace concept to high level semantics of the Android Framework, we carry forward the efficiency advantages of the namespace concept. Just as Linux Namespaces share a single kernel and OS instance, our namespaces share a single Framework and runtime instance. In the constrained environment of a mobile device, this is an extremely important trait.

Trait 4: *Namespaces share by default*. Namespaces provide strong isolation of certain resources while permitting sharing of others [70][68][67] . This contrasts with isolation based on coarse-grained platform or application virtualization where communication among processes in different virtual machines is difficult, and may require modifications to either applications or the virtual machines themselves.

Value to our work: Android is a platform designed around the fact that apps are closely related and must communicate in order to provide the best usability and efficiency. In fact, most of the Android system itself uses these same channels for system-app and system-system communications. Isolating resources by way of fine-grained namespace virtualization at the Framework level preserves continued sharing of resources unrelated to the isolated resource. Coarse-grained virtualization breaks many shared resources that have nothing to do with the isolated resource.

Trait 5: *Namespaces are transparent*. Namespaces are transparent to the host system, as well as to the applications inside them [71][68]. This means that processes running in a namespace appear as normal processes to the host system, and the host system appears normal to an application. The host system retains the full ability to monitor, analyze, and directly control all processes, and the application need not be modified to run in the namespace. This is not always the case with other forms of virtualization such as platform and application virtualization. In the former, the host system loses visibility and control over the individual processes running within the VM, and in the latter, the application may need to be modified to account for unrepresented resources.

Value to our work: By applying the namespace concept to the Framework, a single Android system and kernel retains full control over all applications, and apps do not need modifications to work properly.

Trait 6: *Namespaces have a small footprint* Unlike many forms of platform and application visualization, namespaces do not require persistent files or large binaries, and have a small kernel footprint, making them lightweight [71][70][68][67].

Value to our work: Mobile devices are resource-constrained and most users will become very impatient with security features that consume storage or negatively impact performance. By addressing only a specific, high-level resource, namespaces applied to the Framework can be implemented with relatively few modifications to the Android system, and only a small amount of memory consumed. This results in nearly zero impact on storage resources and device performance. In addition, since mobile devices are usually subject to disadvantaged or expensive communication links, the small footprint opens the possibility of practical over-the-air (OTA) configuration or update.

Appendix K

Android `servicemanager` hypovisor code

This appendix contains the code added to **`frameworks/native/cmds/servicemanager/service_manager.c`** to instantiate the *SystemServices* hypovisor within the native **`servicemanager`** process. The code is explained in Section 5.4.2.

Listing K.1: Modified `do_find_service()` function.

```
// Structure to hold records read from /etc/nspolicy.
struct nspace {
        unsigned uid;
        const char service[32];
        const char modifier[1];
};
struct nspace;

uint32_t do_find_service(struct binder_state *bs, const uint16_t *s, size_t len, uid_t uid,
    pid_t spid)
{
    struct svcinfo *si;

    if (!svc_can_find(s, len, spid)) {
        ALOGE("find_service('%s') uid=%d - PERMISSION DENIED\n",
             str8(s, len), uid);
        return 0;
    }

    /*****************************************************************************
     * RATAZZI 7/14/2014
     *
     * Before calling find_svc(), we check /etc/nspolicy to see if this uid
     * belongs to a namespace for the service requested. If so, we append the the
     * namespace identifier to the service name and increase len appropriately.
     *
     */

    uint16_t *ns_val, *ns_s;
    unsigned int n;

    FILE *fd;
    // Currently, we support two places for nspolicy to live: a system file in /etc, and a file controlled
    // by CSRG's SecureLauncher in that app's data directory. The latter supercedes the former if present.
    char filename1[128] = "/data/data/com.syr.csrg.seclauncher/files/nspolicy";
    char filename2[128] = "/etc/ns/nspolicy";
    char filename[128] = "";
    unsigned int nfields = 3; // <uid> <service_name> <modifier>
    unsigned int lineno = 0;
    int nret;
    struct nspace nsp[128];
    bool nflag = false;

    fd=fopen(filename1, "r");
    if (fd == NULL) {
        fd=fopen(filename2, "r");
        if (fd == NULL) {
            ALOGE("Unable to open %s or %s policy file: %d (%s)\n",
                         filename1, filename2, errno, strerror(errno));
        } else {
            strcpy(filename, filename2);
        }
    } else {
            strcpy(filename, filename1);
    }

    if (fd != NULL) {
        while(!feof(fd)) {
              ++lineno;
          while(nfields == (nret = fscanf(fd, "%d %s %s", &nsp[lineno].uid,
                                                          &nsp[lineno].service,
                                                          &nsp[lineno].modifier))) {
                    if ((uid == nsp[lineno].uid || (uid > 10000 && nsp[lineno].uid == 99999)
                        )
                         && str16eq(s, &(nsp[lineno].service))) {
                       ALOGI("MATCH in do_find_service('%s' requested by uid/pid=%d/%d).
                           Adding '_%s'",
                                    str8(s, len), uid, spid, &nsp[lineno].modifier);
                       nflag = true;
                       len+=2;
                       ns_s = add_ns(s, &(nsp[lineno].modifier));
                       s = ns_s;
                    }
                    ++lineno;
              }
              if(ferror(fd)) {
                 break;
              } else if (nret != EOF) {
                 ALOGE("Ignoring malformed line %d in %s\n", lineno, &filename);
                 fscanf(fd, "%*[^\n]");
              }
        }
        fclose(fd);
    }

    si = find_svc(s, len);

    if (nflag) ALOGI("check_service('%s') handle = %x\n", str8(s, len), si ? si->handle : 0)
        ;
    if (si && si->handle) {
        if (!si->allow_isolated) {
            // If this service doesn't allow access from isolated processes,
            // then check the uid to see if it is isolated.
            uid_t appid = uid % AID_USER;
            if (appid >= AID_ISOLATED_START && appid <= AID_ISOLATED_END) {
                return 0;
            }
        }
        return si->handle;
    } else {
        return 0;
    }
}
```

Listing K.2: ***add_ns()*** *function.*

```
/***********************************************************************************
 * RATAZZI 7/15/2014
 *
 * Function to concatenate the namespace value from namespace[] to the name in
 * the original request.
 *
 * Pointers made this a royal pain and I was forced to use only 1 char for the
 * namespace.  I guess this is OK, 'cause even with only 1 char, we can have a
 * lot of namespaces for each service. Maybe 36 or more, not counting special
 * chars (are they allowed in service names??).
 */

const char *add_ns(uint16_t *x, uint16_t *ns)
{
    static char buf[128];
    unsigned max = 127;
    uint16_t *p=buf;

    if (x) {
        while (*x && max--) {
            *p++ = *x++;
        }
        *p++ = 0x5f; // "_"
        *p++ = *ns++;
    }
    *p++ = 0;
    return buf;
}
```

References

[1] Gartner, Inc. *Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments Are On Pace to Grow 6.9 Percent in 2014*. URL: http://www.gartner.com/newsroom/id/2692318.

[2] *Google, Inc. Q2 2013 Earnings Conference Call Transcript*. URL: http://www.nasdaq.com/aspx/call-transcript.aspx?StoryId=1557292.

[3] *Out of Pocket: A Comprehensive Mobile Threat Assessment of 7 Million iOS and Android Apps*. Tech. rep. FireEye, Inc., Feb. 2015. URL: https://www2.fireeye.com/rs/fireye/images/rpt-mobile-threat-assessment.pdf.

[4] Pulse Secure Mobile Threat Center. *2015 Mobile Threat Report*. Tech. rep. Pulse Secure LLC, 2015. URL: https://www.pulsesecure.net/download/pages/2819/PulseSecure_MobilityReport.pdf.

[5] *Google Android: Vulnerability Statistics*. [Online; accessed 26 Aug 2016]. URL: http://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224.

[6] Erika Chin, Adrienne Porter Felt, Vyas Sekar, and David Wagner. "Measuring User Confidence in Smartphone Security and Privacy". In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. SOUPS '12. Washington, D.C.: ACM, 2012, 1:1–1:16.

[7] *Security Enhancements | Android Open Source Project*. [Online; accessed 9 May 2016]. URL: https://source.android.com/security/enhancements/.

[8] William Enck, Machigar Ongtang, and Patrick McDaniel. *Mitigating Android Software Misuse*

Before It Happens. Tech. rep. NAS-TR-0094-2008. The Pennsylvania State University, Nov. 21, 2008.

[9] Peter Eckersley. *Google Removes Vital Privacy Feature From Android, Claiming Its Release Was Accidental*. Dec. 2013. URL: https://www.eff.org/deeplinks/2013/12/google-removes-vital-privacy-features-android-shortly-after-adding-them.

[10] *Security Overview | Android Open Source Project*. [Online; accessed 9 May 2016]. URL: https://source.android.com/security/.

[11] Gary McGraw and Edward W Felten. *Securing Java: getting down to business with mobile code*. John Wiley & Sons, Inc., 1999.

[12] Samuel Gibbs. "Why it took us so long to match Apple on privacy –a Google exec explains". In: *theguardian* (June 2015). [Online; accessed 12 May 2016]. URL: https://www.theguardian.com/technology/2015/jun/09/google-privacy-apple-android-lockheimer-security-app-ops/.

[13] *Security Enhancements in Android 6.0 | Android Open Source Project*. [Online; accessed 16 May 2016]. URL: https://source.android.com/security/enhancements/enhancements60.html.

[14] *anti-virus - Android Apps on Google Play*. [Online; accessed 16 May 2016]. URL: https://play.google.com/store/search?q=anti-virus&c=apps.

[15] *Android Has a Big Security Problem, But Antivirus Apps Can't Do Much to Help*. [Online; accessed 8 June 2016]. URL: http://www.howtogeek.com/232436/android-has-a-big-security-problem-but-antivirus-apps-cant-do-much/.

[16] Heqing Huang, Kai Chen, Chuangang Ren, Peng Liu, Sencun Zhu, and Dinghao Wu. "Towards Discovering and Understanding Unexpected Hazards in Tailoring Antivirus Software for Android". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. Singapore, Republic of Singapore: ACM, 2015, pp. 7–18. URL: http://doi.acm.org/10.1145/2714576.2714589.

[17] Black Duck Software, Inc. *The Android Open Source Project on Open Hub*. URL:

https://www.openhub.net/p/android (visited on 05/07/2016).

[18] Chee-Sing Chan. “Complexity the Worst Enemy of Security”. In: *Computerworld, Inc.* (Dec. 2012). [Online; accessed 9 May 2016]. URL: http://www.computerworld.com/article/2493938/cyberwarfare/complexity-the-worst-enemy-of-security.html.

[19] Daniel R. Thomas, Alastair R. Beresford, and Andrew 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. Denver, Colorado, USA: ACM, 2015, pp. 87–98.

[20] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. “Android Permissions Demystified”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS ’11. Chicago, Illinois, USA: ACM, 2011, pp. 627–638.

[21] Laurent Simon and Ross Anderson. “PIN Skimmer: Inferring PINs Through the Camera and Microphone”. In: *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. SPSM ’13. Berlin, Germany: ACM, 2013, pp. 67–78.

[22] Liang Cai and Hao Chen. “TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion”. In: *Proceedings of the 6th USENIX Conference on Hot Topics in Security*. HotSec’11. San Francisco, CA: USENIX Association, 2011, pp. 9–9.

[23] Zhi Xu, Kun Bai, and Sencun 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. Tucson, Arizona, USA: ACM, 2012, pp. 113–124.

[24] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. “Tapprints: Your Finger Taps Have Fingerprints”. In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’12. Low Wood Bay, Lake District, UK: ACM, 2012, pp. 323–336.

[25] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. “ACCessory: Password Inference Using Accelerometers on Smartphones”. In: *Proceedings of the Twelfth Workshop on Mobile Computing Systems &#38; Applications*. HotMobile ’12. San Diego,

California: ACM, 2012, 9:1–9:6.

[26] *Manifest.permission | Android Developers*. URL: https://developer.android.com/reference/android/Manifest.permission.html.

[27] Zhi Xu and Jen Miller-Osborn. *Bad Certificate Management in Google Play Store*. [Online; accessed 8 Jun 2016]. Aug. 2014. URL: http://researchcenter.paloaltonetworks.com/2014/08/bad-certificate-management-google-play-store/.

[28] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng 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. San Antonio, Texas, USA: ACM, 2012, pp. 317–326. URL: http://doi.acm.org/10.1145/2133601.2133640.

[29] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. "Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, California, USA*. 2016.

[30] *Dashboards | Android Developers*. [Online; accessed 7 July 2016]. URL: https://developer.android.com/about/dashboards/.

[31] *Quick Look at Android Lollipop's Managed Provisioning*. [Online; accessed 11 June 2016]. URL: https://epratazzi.wordpress.com/2014/07/24/android-l-managed-profile-quick-look/.

[32] *Android 4.2 APIs | Android Developers*. URL: http://developer.android.com/about/versions/android-4.2.html#MultipleUsers.

[33] Amit Singh. *A Taste of Computer Security*. [Online; accessed 12 Aug 2016]. Aug. 2014. URL: http://www.kernelthread.com/publications/security/uw.html.

[34] Paul Ratazzi, Yousra Aafer, Amit Ahlawat, Hao Hao, Yifei Wang, and Wenliang Du. "A Systematic Security Evaluation of Android's Multi-User Framework". In: *Mobile Security Technologies (MoST) 2014*. MoST'14. San Jose, CA, USA, May 2014.

[35] *External Storage Technical Information | Android Developers*. URL:

http://source.android.com/devices/tech/storage/.

[36] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly Media, Inc., 2013.

[37] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. "A Study of Android Application Security". In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, pp. 21–21.

[38] *The Android Source Code*. URL: http://source.android.com/source/.

[39] *Settings | Android Developers*. URL: http://developer.android.com/reference/android/provider/Settings.html.

[40] *<activity> | Android Developers*. URL: http://developer.android.com/guide/topics/manifest/activity-element.html#exported.

[41] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. "Systematic detection of capability leaks in stock Android smartphones". In: *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*. 2012.

[42] Bruce Schneier. "A Plea for Simplicity: You can't secure what you don't understand". In: *Schneier on Security* (Nov. 1999). [Online; accessed 12 May 2016]. URL: https://www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html.

[43] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael 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. Denver, Colorado, USA: ACM, 2015, pp. 1248–1259. URL: http://doi.acm.org/10.1145/2810103.2813648.

[44] Roberto Gallo, Patricia Hongo, Ricardo Dahab, Luiz C. Navarro, Henrique Kawakami, Kaio Galvão, Glauber Junqueira, and Luander 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, New York: ACM,

2015, 12:1–12:6. URL: http://doi.acm.org/10.1145/2766498.2766519.

[45] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. “The Impact of Vendor Customizations on Android Security”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*. CCS ’13. Berlin, Germany: ACM, 2013, pp. 623–634. URL: http://doi.acm.org/10.1145/2508859.2516728.

[46] Yousra Aafer, Xiao Zhang, and Wenliang Du. “Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 1153–1168. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aafer.

[47] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. “Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework”. In: *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS’16). ISOC*. 2016.

[48] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. “Checking System Rules Using System-specific, Programmer-written Compiler Extensions”. In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI’00. San Diego, California: USENIX Association, 2000. URL: http://dl.acm.org/citation.cfm?id=1251229.1251230.

[49] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An Empirical Study of Operating Systems Errors”. In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP ’01. Banff, Alberta, Canada: ACM, 2001, pp. 73–88. URL: http://doi.acm.org/10.1145/502034.502042.

[50] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. “Faults in Linux: Ten Years Later”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 305–318. URL: http://doi.acm.org/10.1145/1950365.1950401.

[51] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Dead or Alive: Finding Zombie Features in the Linux Kernel". In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. FOSD '09. Denver, Colorado, USA: ACM, 2009, pp. 81–86. URL: http://doi.acm.org/10.1145/1629716.1629732.

[52] Andreas Ziegler, Valentin Rothberg, and Daniel Lohmann. "Analyzing the Impact of Feature Changes in Linux". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 25–32. URL: http://doi.acm.org/10.1145/2866614.2866618.

[53] Stephen Smalley and Robert Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android". In: *Proceedings of the 2013 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, Feb. 2013.

[54] *<permission> | Android Developers*. [Online; accessed 20 Sep 2016]. URL: https://developer.android.com/guide/topics/manifest/permission-element.html#plevel.

[55] Wikipedia. *Feature Engineering*. [Online; accessed 23 Sep 2016]. 2016. URL: https://en.wikipedia.org/wiki/Feature_engineering.

[56] Pedro Domingos. "A Few Useful Things to Know About Machine Learning". In: *Commun. ACM* 55.10 (Oct. 2012), pp. 78–87. URL: http://doi.acm.org/10.1145/2347736.2347755.

[57] Amit Ahlawat. *FeatureExtraction (v4.0, "Atlantis")*. [Unpublished software]. Syracuse University, Syracuse, NY. Apr. 2016.

[58] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. "Linux Security Modules: General Security Support for the Linux Kernel". In: *USENIX Security Symposium*. Vol. 2. 2002, pp. 1–14.

[59] J. G. R. Sathiaseelan, S. Albert Rabara, and J. Ronal Martin. "Multi-Level Secure Framework (MLSF) for Composite Web Services". In: *Proceedings of the 2Nd International Conference on*

Interaction Sciences: Information Technology, Culture and Human. ICIS '09. Seoul, Korea: ACM, 2009, pp. 580–585. URL: http://doi.acm.org/10.1145/1655925.1656030.

[60] Maciej P. Machulak, Łukasz Moreń, and Aad van Moorsel. "Design and Implementation of User-managed Access Framework for Web 2.0 Applications". In: *Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing*. MW4SOC '10. Bangalore, India: ACM, 2010, pp. 1–6. URL: http://doi.acm.org/10.1145/1890912.1890913.

[61] *Binder | Android Developers*. [Online; accessed 9 May 2016]. URL: http://developer.android.com/reference/android/os/Binder.html.

[62] *Android's First Multi Boot Application: init2winitapps/BootManager*. URL: https://github.com/init2winitapps/BootManager.

[63] Christoffer Dall and Jason Nieh. "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 333–348.

[64] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. "AirBag: Boosting Smartphone Resistance to Malware Infection". In: *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*. 2014.

[65] Christoffer Dall, Jeremy Andrus, Alexander Van't Hof, Oren Laadan, and Jason Nieh. "The Design, Implementation, and Evaluation of Cells: A Virtual Smartphone Architecture". In: *ACM Trans. Comput. Syst.* 30.3 (Aug. 2012), 9:1–9:31.

[66] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Practical and Lightweight Domain Isolation on Android". In: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011, pp. 51–62.

[67] Eric Biederman. "Multiple Instances of the Global Linux Namespaces". In: *Ottawa Linux Symposium (OLS) 2006*. Ottawa, ON, CAN, Aug. 2006.

[68] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. “Virtual Servers and Checkpoint/Restart in Mainstream Linux”. In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 104–113.

[69] Rami Rosen. “Linux Containers and the Future Cloud”. In: *Linux J.* 2014.240 (Apr. 2014).

[70] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. “Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors”. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys ’07. Lisbon, Portugal: ACM, 2007, pp. 275–287.

[71] David Strauss. “The Future Cloud is Container, Not Virtual Machines”. In: *Linux J.* 2013.228 (Apr. 2013).

[72] Wikipedia. *Supervisory program*. [Online; accessed 24 Aug 2016]. 2016. URL: https://en.wikipedia.org/wiki/Supervisory_program.

[73] Wikipedia. *IBM System/360 Model 67*. [Online; accessed 24 Aug 2016]. 2016. URL: https://en.wikipedia.org/wiki/IBM_System/360_Model_67.

[74] Neelima Krishnan. “Android Hypovisors: Securing Mobile Devices through High-Performance, Light-Weight, Subsystem Isolation with Integrity Checking and Auditing Capabilities”. MA thesis. Virginia Tech, Dec. 2014.

[75] Xiangyu Liu, Wenrui Diao, Zhe Zhou, Zhou Li, and Kehuan Zhang. “Gateless Treasure: How to Get Sensitive Information from Unprotected External Storage on Android Phones”. In: *CoRR* abs/1407.5410 (2014).

[76] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. “Permission Re-delegation: Attacks and Defenses”. In: *Proceedings of the 20th USENIX Conference on Security*. SEC’11. San Francisco, CA: USENIX Association, 2011, pp. 22–22.

[77] X. Zhang and W. Du. “Attacks on Android Clipboard”. In: *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Egham, UK, July 2014.

[78] Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. "A11Y Attacks: Exploiting Accessibility in Operating Systems". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 103–115.

[79] Zhi Xu and Sencuno Zhu. "Abusing Notification Services on Smartphones for Phishing and Spamming". In: *Proceedings of the 6th USENIX Workshop on Offensive Technologies*. Bellevue, WA: USENIX, 2012.

[80] Yves-Alexandre de Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. "Unique in the Crowd: The privacy bounds of human mobility". In: *Scientific reports* 3 (2013).

[81] F. Mohsen and M. Shehab. "Android keylogging threat". In: *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*. Oct. 2013, pp. 545–552.

[82] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. "Practicality of Accelerometer Side Channels on Smartphones". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida: ACM, 2012, pp. 41–50.

[83] Lingguang Lei, Yuewu Wang, Jian Zhou, Daren Zha, and Zhongwen Zhang. "A Threat to Mobile Cyber-Physical Systems: Sensor-Based Privacy Theft Attacks on Android Smartphones". In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*. July 2013, pp. 126–133.

[84] Keisuke Komeda, Masahiro Mochizuki, and Nobuhiko Nishiko. "User Activity Recognition Method Based on Atmospheric Pressure Sensing". In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*. UbiComp '14 Adjunct. Seattle, Washington: ACM, 2014, pp. 737–746.

[85] *Overview of Google Play Services*. [Online; accessed 24 Aug 2016]. URL: https://developers.google.com/android/guides/overview.

[86] Mohammad Mannan and P.C. van Oorschot. "Leveraging personal devices for stronger

password authentication from untrusted computers". In: *Journal of Computer Security* 19.4 (Jan. 2011), pp. 703–750.

[87] Dianne Hackborn. *Re: [PATCH 1/6] staging: android: binder: Remove some funny && usage*. June 2009. URL: https://lkml.org/lkml/2009/6/25/3.

[88] *InputMethodManager | Android Developers*. URL: http://developer.android.com/reference/android/view/inputmethod/InputMethodManager.html.

[89] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards taming privilege-escalation attacks on Android". In: *19th Annual Network & Distributed System Security Symposium (NDSS)*. Vol. 17. 2012, pp. 18–25.

[90] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. "Xmandroid: A new Android evolution to mitigate privilege escalation attacks". In: *Technische Universität Darmstadt, Technical Report TR-2011-04* (2011).

[91] Yajin Zhou and Xuxian Jiang. "Detecting passive content leaks and pollution in Android applications". In: *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*. 2013.

[92] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. "QUIRE: Lightweight Provenance for Smart Phone Operating Systems." In: *USENIX Security Symposium*. 2011.

[93] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. "Permission Re-Delegation: Attacks and Defenses." In: *USENIX Security Symposium*. 2011.

[94] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies". In: *22nd USENIX Security Symposium (USENIX Security'13). USENIX*. 2013.

[95] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. "Chex: statically vetting Android apps for component hijacking vulnerabilities". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 229–240.

[96] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. "Droidchecker: analyzing Android

applications for capability leak”. In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM. 2012, pp. 125–136.

[97] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. “Activity Recognition Using Cell Phone Accelerometers”. In: *SIGKDD Explor. Newsl.* 12.2 (Mar. 2011), pp. 74–82.

[98] Emiliano Miluzzo, Nicholas D. Lane, Kristóf Fodor, Ronald Peterson, Hong Lu, Mirco Musolesi, Shane B. Eisenman, Xiao Zheng, and Andrew T. Campbell. “Sensing Meets Mobile Social Networks: The Design, Implementation and Evaluation of the CenceMe Application”. In: *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*. SenSys ’08. Raleigh, NC, USA: ACM, 2008, pp. 337–350.

[99] Young-Seol Lee and Sung-Bae Cho. “Activity Recognition Using Hierarchical Hidden Markov Models on a Smartphone with 3D Accelerometer”. In: *Proceedings of the 6th International Conference on Hybrid Artificial Intelligent Systems - Volume Part I*. HAIS’11. Wroclaw, Poland: Springer-Verlag, 2011, pp. 460–467.

[100] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L. Reyes-Ortiz. “Human Activity Recognition on Smartphones Using a Multiclass Hardware-friendly Support Vector Machine”. In: *Proceedings of the 4th International Conference on Ambient Assisted Living and Home Care*. IWAAL’12. Vitoria-Gasteiz, Spain: Springer-Verlag, 2012, pp. 216–223.

[101] Felix Rohrer, Yuting Zhang, Lou Chitkushev, and Tanya Zlateva. “DR BACA: Dynamic Role Based Access Control for Android”. In: *Proceedings of the 29th Annual Computer Security Applications Conference*. ACSAC ’13. New Orleans, Louisiana: ACM, 2013, pp. 299–308.

[102] Zemin Liu, Choon-Sung Nam, and Dong-Ryeol Shin. “UAMDroid: A user authority manager model for the Android platform”. In: *Advanced Communication Technology (ICACT), 2011 13th International Conference on*. Feb. 2011, pp. 1146–1150.

[103] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. “TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *Proceedings of the 9th USENIX*

Conference on Operating Systems Design and Implementation. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–6.

[104] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David 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. Chicago, Illinois, USA: ACM, 2011, pp. 639–652.

[105] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. "Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications". In: *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 3–14.

[106] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints". In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '10. Beijing, China: ACM, 2010, pp. 328–332.

[107] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. "Compac: Enforce Component-level Access Control in Android". In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY '14. San Antonio, Texas, USA: ACM, 2014, pp. 25–36.

[108] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. "MOSES: Supporting Operation Modes on Smartphones". In: *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. SACMAT '12. Newark, New Jersey, USA: ACM, 2012, pp. 3–12.

[109] W. Chen, L. Xu, G. Li, and Y. Xiang. "A Lightweight Virtualization Solution for Android Devices". In: *Computers, IEEE Transactions on* PP.99 (2015), pp. 1–1.

[110] Kassem Fawaz and Kang G. Shin. "Location Privacy Protection for Smartphone Users". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 239–250.

[111] Zhigang Chen, Xin Hu, Xiaoen Ju, and K.G. Shin. "LISA: Location Information ScrAmbler for privacy protection on smartphones". In: *Communications and Network Security (CNS), 2013 IEEE Conference on*. Oct. 2013, pp. 296–304.

[112] Saikat Guha, Mudit Jain, and Venkata N. Padmanabhan. "Koi: A Location-privacy Platform for Smartphone Apps". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 14–14.

[113] *Security program overview*. [Online; accessed 2 Nov 2016]. URL: https://source.android.com/security/#security-program-overview.

[114] *System and kernel security | Android Open Source Project*. [Online; accessed 2 Nov 2016]. URL: https://source.android.com/security/overview/kernel-security.html.

[115] *Application security | Android Open Source Project*. [Online; accessed 2 Nov 2016]. URL: https://source.android.com/security/overview/app-security.html.

[116] *Security updates and resources | Android Open Source Project*. URL: https://source.android.com/security/overview/updates-resources.html.

[117] *Security Enhancements in Android 5.0 | Android Open Source Project*. [Online; accessed 16 May 2016]. URL: https://source.android.com/security/enhancements/enhancements50.html.

[118] *Security Enhancements in Android 4.4 | Android Open Source Project*. [Online; accessed 16 May 2016]. URL: https://source.android.com/security/enhancements/enhancements44.html.

[119] *Security Enhancements in Android 4.3 | Android Open Source Project*. [Online; accessed 16 May 2016]. URL: https://source.android.com/security/enhancements/enhancements43.html.

[120] *Security Enhancements in Android 4.2 | Android Open Source Project*. [Online; accessed 16 May 2016]. URL: https://source.android.com/security/enhancements/enhancements42.html.

[121] *Security Enhancements in Android 1.5 through 4.1 | Android Open Source Project*. [Online; accessed 16 May 2016]. URL: https://source.android.com/security/enhancements/enhancements41.html.

[122] Ben Gruver. *Deodex Instructions*. [Online; accessed 15 Dec 2015]. Oct. 2015. URL: https://github.com/JesusFreke/smali/wiki/DeodexInstructions.

[123] joeldroid. *JoelDroid Lollipop Batch Deodexer V 2.5*. [Online; accessed 15 Dec 2015]. Apr. 2015. URL: http://forum.xda-developers.com/android/software-hacking/script-app-joeldroid-lollipop-batch-t2980857.

[124] matt95. *Unruu HTC RUUs*. [Online; accessed 15 Dec 2015]. May 2013. URL: http://forum.xda-developers.com/showthread.php?t=2264238.

[125] joeykrim. *RUUVEAL - Decrypt the `rom.zip` file included in the RUU*. [Online; accessed 15 Dec 2015]. Jan. 2013. URL: http://forum.xda-developers.com/showthread.php?t=2084470.

[126] Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E. Porter. "Practical Techniques to Obviate Setuid-to-root Binaries". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 8:1–8:14.

[127] Michael Kerrick. "Namespaces in operation". In: *LWN.net* (). [Online; accessed xx]. URL: http://lwn.net/Articles/531114/.

[128] Wikipedia. *Namespace*. [Online; accessed 12 Aug 2015]. 2015. URL: http://en.wikipedia.org/wiki/Namespace.

[129] Rami Rosen. "Linux Kernel Networking: Implementation and Theory". In: New York: Apress, 2014. Chap. 14, pp. 405–426.

Vita

Author’s Name:	Edward Paul Ratazzi
Place of Birth:	Fort Huachuca, Arizona, USA
Date of Birth:	August 22, 1965

Degrees Awarded:

Bachelor of Science in Electrical Engineering, Rensselaer Polytechnic Institute, 1987

Master of Science in Electrical Engineering, Syracuse University, 1992

Master of Science in Management, Rensselaer Polytechnic Institute, 2006

Professional Experience:

Principal Engineer, Air Force Research Laboratory Information Directorate, July 2006-present

Adjunct Instructor, Syracuse University, February 2005-December 2013

Senior Electronics Engineer, AFRL, November 1997-July 2006

Electronics Engineer, Rome Laboratory, January 1992-November 1997

Electronics Engineer, Rome Air Development Center, June 1987-January 1992

Software Intern, Measurement Concept Corporation, January 1980-January 1981