UNIVERSAL SKEPTIC BINDER-DROID - TOWARDS ARRESTING MALICIOUS COMMUNICATION OF COLLUDING APPS IN ANDROID

A Thesis

by

SRINATH NADIMPALLI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Guofei Gu
Committee Members,	Ricardo Bettati
	A. L. Narasimha Reddy
Head of Department,	Dilma Da Silva

May 2015

Major Subject: Computer Science

ABSTRACT

Since its first release, Android has been increasingly adopted by people and companies worldwide. It is currently estimated that around 1.1 billion Android devices are in use. Even though Android was built with Security principles and comes with a sound security model, it is a favorite target for malware authors. McAfee observed a 76% year on year growth in Android malware during the year 2014 alone. Thus malware is a predominant threat in Android ecosystem. A common attack vector for Android malware is the use of colluding apps. Colluding apps involve two or more applications and operate in two phases. In Phase 1, one application steals private sensitive data of the user In Phase 2, the same application sends the data to another application via covert communication channels. There are several covert channels in Android frameworks. Until now, several solutions in the literature have focused on preventing the extraction of sensitive data from the phone. We, to the best of our knowledge, are the first to stop the flow of sensitive info via the covert channels. We propose, *Universal Skeptic Binder-Droid*, an enhanced Binder module which enforces policies regarding the use of communication channels and prevents apps from colluding. With our proposed system, we have the added advantage of dynamically configuring policies at run time. Our initial implementation and results on our test bed reflect on the effectiveness and the ease of use of such a system.

DEDICATION

Dedicated to my loving and caring mother, father and sister...

No words can express my gratitude for your never ending love, support, kindness and encouragement.

ACKNOWLEDGEMENTS

During my Master's thesis, I have been very lucky in receiving a lot of feedback, input and encouragement from a lot of people. This work would not have been possible had it not been for all of them.

Foremost, I would like to thank my advisor, Dr. Guofei Gu, for his consistent support and guidance. Dr. Gu guided me towards the best direction, challenged me in my thinking process, molded and honed my reasoning capabilities. I am forever indebted to him for improving my reasoning skills and strengthening my ability to objectively analyze concepts. I thank him profusely for making it increasingly challenging to defend my ideas day after day, week after week. To say that getting his approval for an idea is the biggest validation we seek is a gross understatement. I hope I can influence people in the future as much as he has influenced me.

Secondly, I would like to thank my committee members, Dr. Ricardo Bettati and Dr. A. L. Narasimha Reddy for their interest in my work. I also thank them for their insightful comments that has significantly improved the quality of my work.

I have been very fortunate to be a part of the talented SUCCESS lab group. I thank Guangliang Yang, Kevin Hong, Jialong Zhang, Haopei Wang, Lei Xu, Robert Baykov, Abner Mendoza, Xu Yan, Visvanathan Thothathri and David Cox for their valuable company. I thank Guangliang Yang for his assistance on my thesis and I thank Zhaoyan Xu and several of my lab mates for the feedback they provided throughout my Masters. Special thanks to Robert Baykov and Abner Mendoza, the lab-mates I share my workspace with, for all the small talk, moments of laughter, hope, happiness and for being my reality check with their extremely honest feedback.

It would be a great miss on my part if I do not thank the amazing friends at

the Indian Graduate Students Association (IGSA). I've had the opportunity to meet several amazing people during my term there and I am proud to be a part of the premier organization serving Indian Aggies at Texas A&M.

Last but not the least, I thank my beloved parents and my sister for all their support and encouragement. I owe them my life. Its not possible to describe their unflinching support and encouragement for me and my dreams. I would not have become what I am today had it not been for them.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Android is a Linux based Operating System built for hand-held devices (smartphones, tablets) by Android Inc. before being acquired by Google in 2005. In addition to developing the OS, Android Inc. also built handsets for their new OS [1], [2]. In 2007, Google along with the collaboration of many other technological and telecommunication companies formed the *Open Handset Alliance* (OHA). OHA's goal was to build the first truly open and comprehensive platform for mobile devices [3]. The first commercial release of Android came a year later on 23rd September 2008. Along with the commercial release, Google announced the launch of Android Market (later renamed to *Google Play Store*), an on-line marketplace where users could install new applications and updates to existing applications. Thus, an entire ecosystem comprising the marketplace, Android OS and hardware devices was created.

Ever since its first release, the popularity of Android powered devices increased rapidly. New members collaborated with the OHA consortium and contributed to growth of the platform. Android's user-base increased quickly over time. It is currently estimated that around 1.5 million Android devices are activated every day [4]. Gartner [5] reports that approximately 0.8 billion Android powered devices were sold in 2013. It was also estimated that sales of Android phones will reach 1 billion in 2014 [6].

This widespread adoption made Android a very attractive market for hobbyists, independent developers, start-up companies and well-established companies (to monetize). The open nature of Android system made it very easy to launch applications into the marketplace. It is currently estimated that over 1 million applications exist

in the Play Store [7] and these apps have been downloaded over 50 billion times [4]. These statistics make us infer that an average user has downloaded, installed, used and uninstalled around 50 applications.

Many apps store personal information of the owner - information ranging from his/her personal contacts to sensitive bank account details on the device. Given the huge user-base, the abundance of apps available for use, and the sensitivity of data stored by the apps, it is only natural that malware authors exploit existing security issues in Android ecosystem for their own benefit.

Malware authors have released malware that steals private user information from the phone or cost the users monetary losses by sending SMS to premium numbers. In a 2013 report [8], McAfee noted that around 17000 new Android malware were found in the second quarter of 2013. In a recent report by McAfee [9], McAfee labs observed the year-over-year growth of 76 percentage in mobile malware. These clearly indicate that Android devices are facing an ever increasing threat of malware.

Zhou et al. [10] have studied and characterized Android malware and the commonly used attack vectors. Common attack vector used is to repackage malware with other benign apps. The repackaged app is then installed on an unsuspecting user device via a social engineering attack or fake marketplaces. The malware in the repackaged app infects the phone/collects sensitive private data from the phone/user. However to prevent suspicion and/or to elongate the attack vector, the malware component communicates with another application on the phone to transmit the sensitive data out of the phone. This is how repackaged apps work in achieving malicious ends on Android powered devices.

To transmit sensitive data to the recipient app, the malware component can use both overt and covert communication channels. An overt channel is a medium through which two entities communicate via conventional methods (such as Shared-

Preferences, Intents, Internal/External storage). A covert channel is a medium through which two entities communicate via non-conventional methods [11], [12]. These channels are inherited by Android via the Linux kernel or arise due to design flaws in the Android framework. Existing work in the literature solves the issue of colluding apps by tracking overt channels for flow/extraction of sensitive data by the use of program modeling, data tainting etc. There has been very little work to address communication via covert channels.

In our work, we address the colluding apps problem by stopping them from communicating via covert channels. To the best of our knowledge, we are the first to develop a comprehensive system for achieving the same. We propose *Universal Skeptic Binder-Droid (USB-Droid)* - an enhancement to the Binder module of Android framework. USB-Droid prevents collusion in covert channels by the use of policy engine in the Binder layer. To make it easy for system administrators to use the system, we design an easy to learn policy language syntax to define new policies.

Our contributions can be summarized as follows:

- Bringing all covert channels under the purview of Binder infrastructure (Binder protocol and tokens, Binder driver and Service manager)
- A novel way of preventing apps from colluding by the use of policy engine in the Binder layer
- An easy to learn policy language to define new policies
- Reference implementation of the proposal with minimal overhead

The rest of the thesis is organized as follows - section 2 provides the required background information for our work, followed by an extensive survey of the Android ecosystem in section 3. We present USB-Droid's design in section 4, implementation

in section 5. We evaluated our proposal in section 6, propose future work and identify limitations in section 7 and provide concluding remarks in section 8.

2. BACKGROUND

2.1 Android Ecosystem - A Primer

In this section, we provide a brief introduction to components within Android ecosystem. This will provide sufficient background information for the reader to understand the rest of the thesis with ease.

2.1.1 Android Marketplaces

Android Marketplaces are on-line/electronic software distribution platforms where users can download new apps and/or download updates to apps already installed on their devices. One reason why Android ecosystem thrives so well is the abundance of apps that are available irrespective of multiple form factors (smart-phones and tablets). These apps are available for download from several marketplaces. Some notable marketplaces are Google's Play Store [13], Amazon Appstore for Android [14], the open source F-Droid [15], Baidu's App Store [16], Anzhi [17], Tencent's App Gem [18]. All marketplaces other than Google's Play Store are called *alternative markets* in the literature. In addition to allowing users to download apps from the official Play Store, Android also allows installation of apps from alternative marketplaces. This is termed as *side-loading*.

Developers make apps available for public download via the Play Store or alternative marketplaces. To publish apps (in the Play Store), one pays a one-time fee to Google. The fees was $25 at the time of this writing. One determines the app's content rating, availability (free/paid) and provides screen-shots and/or videos for the users to "preview" the app before installing it. After submitting the above details, one uploads the installation file (*apk* format). The detailed steps of releasing an app are captured in the launch checklist [19].

2.1.2 Hardware Vendors

Android Operating System is an open source project with Google releasing every new version's source code to the open source community via the Android Open Source Project (AOSP) [20]. Hardware vendors customize the source code released into the AOSP channel with their brand specific changes. It is common for hardware vendors to provide extraneous components and/or apps that come pre-installed with every phone they release.

Hardware vendors also assemble the necessary hardware devices such as System-on-Chip (SoC), Graphics Processing Units (GPU), Random Access Memory (RAM) chips, Read Only Memory (ROM), removable storage (SD-Cards), display unit (LCD or AMOLED), batteries, cameras and several sensors.

2.1.3 Telecommunication Networks

Telecommunication networks, also known as Carrier networks, provide and support the telecommunication infrastructure to Android smart-phones. This includes telephony facilities, messaging services (SMS and MMS) and 2G/3G/4G data transfer facilities.

2.1.4 Android Architecture

The architecture of Android OS is as shown in Fig. 2.1

2.1.4.1 Linux Kernel

The lowermost layer in the system architecture is the Linux kernel. It contains essential hardware drivers that interface with hardware devices such as camera, display, memory, Wi-Fi, speakers, microphone etc... In addition to that, the kernel provides essential system functionality such as memory management, process management and network management. The kernel used in Android is a vanilla open

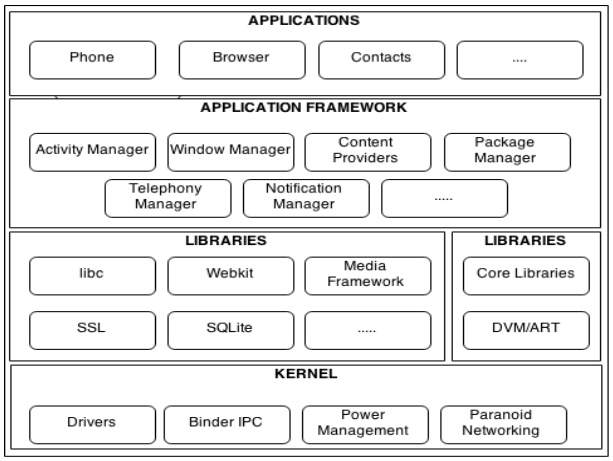

Figure 2.1: Android System Architecture showing the kernel layer, runtime and libraries layer, application framework layer and the applications layer

source version which is adapted for smart-phones and other hand held devices. Some of the significant modifications include

- wakelocks,
- anonymous shared memory (ashmem),
- low memory killer (lowmem) and
- *Binder*

The most significant change is the *Binder* Inter-Process Communication mechanism. Originally derived from the OpenBinder project, Android's Binder module is the basis for all Inter-Process Communication in Android. A kernel driver module, called the Binder driver, facilitates communication between two objects. The communication protocol is a server-client model based on ioctl(). Each Binder object is identified by a token, called Binder token, which allows a binder to be uniquely identified across process boundaries. Thus Binder supports invocation of remote

objects. When a process A wants to communicate with another process B, it uses the Binder for communication. The source (A) first sends the request to the Binder kernel module with details about the destination (B). Binder now has to figure out the identify of the destination. This is achieved by means of ServiceManager module. The ServiceManager module maintains a table of the system's services and their Binder objects. Given a service name, the ServiceManager will return its Binder object if one exists, else create a new Binder object and return its reference. Once B's Binder reference is retrieved by the Binder driver, then it can forward A's request to B. Since Android does not support any of the SysV IPC mechanisms, the Binder is the primary IPC mechanism inside Android. Binder IPC is heavily used inside the framework modules.

The working of the Binder module is depicted in the Figure 2.2

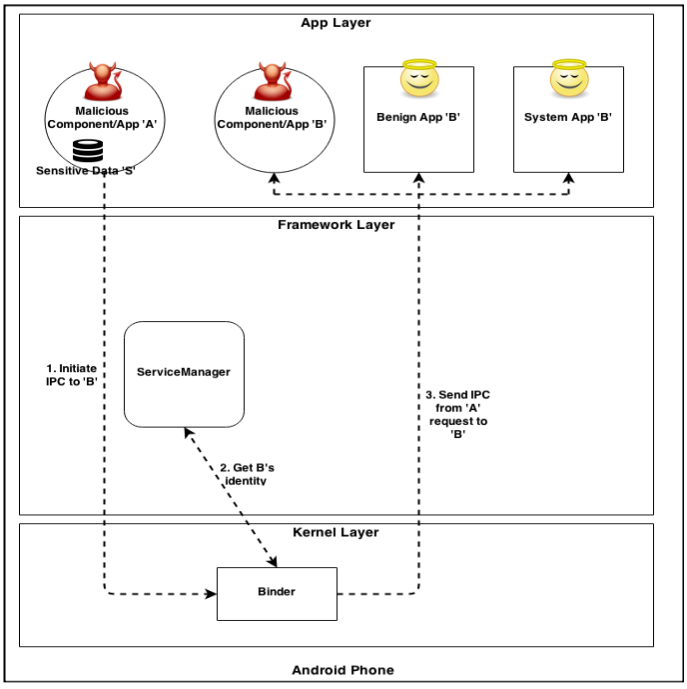

Figure 2.2: Android App Life-Cycle showing different states an app may be during its lifetime

Contrary to desktop workstations, smart-phones and other hand held devices have limited battery life. Hence, the kernel is modified to go to sleep as soon as possible and as frequently as possible. However, to allow apps complete their long-running tasks, the kernel provides *wakelocks*. A wakelock prevents the kernel from sleeping until all acquired wakelocks (by apps) are released.

Anonymous Shared Memory (*ashmem*) facilitates sharing memory between processes and helps reclaim memory under duress. *pmem* allows sharing of physical memory between processes. The kernel employs a *Paranoid Networking* philosophy where the kernel gates all network devices.

2.1.4.2 Libraries

The layer above the Android kernel (in Fig. 2.1) contains several libraries that are imported from third party open source projects. Some of the popular libraries include Webkit (HTML rendering engine), libc (Standard C library), SSL (OpenSSL), SQLite (SQL database engine) and media framework.

2.1.4.3 Android Runtime

Android run-time layer consists of Core libraries and the Dalvik Virtual Machine (DVM). The core libraries are a complementary set of libraries offering the same functionality as that of Java's core libraries. Application code written in Java is translated to Dalvik bytecode and stored in .dex files (Dalvik EXecutable). The generated dex files are then executed by the DVM. DVM is comparable in theory to Java Virtual Machine (JVM) though it differs considerably in practice from JVM. In Android OS 4.4, Google released an alternative run-time called the Android Run-Time (ART) [21]. ART is designed to be faster than DVM and exhibits increased install times for apps due to Ahead of Time compilation. As of Android 5.0, ART runtime replaces DVM completely as the default runtime [22].

2.1.4.4 Android Framework

Above the run-time and the libraries layer (in Fig. 2.1) is the Android framework layer. It serves as a middle-ware between the applications layer and the kernel layer and provides most of Android's unique features. The libraries provided in this layer include Activity Manager, Package Manager, Window Manager, Telephony Manager, Content Providers, Location and Notification Manager.

2.1.4.5 Applications Layer

The top-most layer in the architecture (in Fig. 2.1) is the applications layer. All apps are present in this layer. There are two types of applications namely stock apps and user apps. Stock apps come pre-installed with the phone whereas user apps are available for download from the Play Store and alternative marketplaces. Applications can be built in Java using the SDK or in native code using the Native Development Kit (NDK) [23]. Each app includes a special file called *Android Manifest*. The manifest is an XML file that uniquely identifies the app by its package name. It also declares all the components and permissions the app uses.

2.1.4.5.1 Android Apps Components

The basic building block of an Android app is a component. There are four types of components in Android, namely Activities, Services, Broadcast Receivers and Content Providers. An activity is the user interface shown to a user and is associated with one functionality of the app. Services handle all background processing for an app. Broadcast Receiver handle communication between Android OS and applications. Content Providers facilitate sharing of data between applications.

2.1.4.5.2 Android Apps Execution Life-Cycle

As a user interacts with an app, each activity transitions between four states during its life-cycle:

- Running - This is the state where an activity is running and a user is interacting with it.
- Paused - This is the state where an activity is partially obscured by another activity.
- Stopped - This is the state where an activity is completely obscured by another activity. The activity is no longer running.
- Destroyed - This is the state where a previously Paused/Stopped activity is dropped from memory. Dropping an activity from memory may happen when there is demand to make space for new activities.

2.2 Android Security Model

In this section, we present the security model currently employed in the Android OS. Android's security model is inspired by Linux's security model. We discuss enhancements made by Android security model and present how isolation is achieved by Android. We present how resources are protected by virtue of permissions and identify security trade-offs made in the model.

2.2.1 Separation Of Concerns

Android was designed with security in mind and its model takes after the Linux security model. Android uses the idea of UID to achieve isolation of apps and strengthens it further. In Android, every app is assigned a unique UID and GID

during installation. The app's resources are configured to be accessible by only the UID-GID pair created during installation. Thus, all resources of an app are accessible by the app alone. Owing to the above enhancement and Linux's Separation of Concerns, one app cannot access any other app's data.

However, under legitimate conditions, it may be required that one app uses data available in another app (with possibly different package names). Android OS allows this as an exception to the aforementioned separation of concerns. An application developer can indicate two different apps to share the same UID (*SharedUID*). By virtue of separation of concerns, only these two apps can access each other's data. No other app can access data of either of these two apps. Thus, two or more apps can share data between each other. However, it should be ensured that all the packages are signed using the same digital signature.

2.2.2 Android File-System Isolation

Apps store their data under the */data/data/app_package_name* directory - where app_package_name is the full package name of the app (eg: com.android.example). The permissions of that directory are set as follows: The owner of the directory is set to the current UID. The group and world permissions are not set. Thus, only apps with the same UID can access each other's files. Only a root user (superuser) can access all files.

The above isolation pertains only to the internal file-system storage. External storage media such as SD-cards do not come under the purview of isolation. Thus data written to external storage (SD Cards) lacks permission control. An application developer can overcome this by assigning permissions to files written on external storage. While doing so, it is recommended that he/she adhere to the *Principle of Least Privilege*. The Principle of Least Privilege states that *enough permissions*

should be assigned to do what needs to be done and no more [24].

2.2.3 Android Permissions

Android apps are designed on the "consumer model" paradigm where apps are consumers of data protected by permissions. Each app has to be granted permissions for accessing the data/resources held by the permissions. During installation the user is presented with a list of permissions that the app requests. He/she can grant all permissions or deny all permissions. (*all-or-none*). Currently, Android does not allow granting a subset of the requested permissions.

Allowing the user to make this decision enables him to understand what permissions are required by the app and evaluate the risk in granting the requested permissions. It also assures the user that the app cannot do more than what he/she allowed it to do during installation.

Android OS provides several permissions that are broadly classified under four categories. The categories are arranged in a non-decreasing order of risk:

- *Normal* - these permissions cause no harm to data stored on the phone and are granted to apps by default.
- *Dangerous* - these permissions can cause harm to data and are always shown to the user for review.
- *Signature* - these permissions are granted automatically to a requesting app if it has the same signature as that of the app which created the permission.
- *SignatureorSystem* - these permissions are typically available for the Android team and device manufacturers.

It should be noted that only Normal and Dangerous permissions are available for third party app developers. This largely reduces the attack space since the availabil-

ity of Signature and SignatureorSystem permissions can cause serious consequences if they are misused.

In addition to system defined permissions, it is also possible for app developers to define their own permissions. All permissions are to be declared in the *AndroidManifest.xml* file. This file is packaged along with the app installer (apk file) and is processed by the PackageManager during installation.

2.3 Covert Channels

A covert channel of communication is defined as the medium of communication which has not been designed by the Android designers for communication between two or more communicating bodies. Chandra et al [12] conduct a systematic study of the covert communication channels. These channels have been identified by various work [25, 26, 27, 28] and were systematized by [12]. Based on their systemization, we note down two levels of covert channels - viz OS level and Hardware level. The various overt and covert channels are enumerated below.

Based on our analysis and review of pertinent literature [11],[12], we observed the following properties of colluding channels.

1. There are two communicating parties - Sender and Receiver - and a channel
2. For a covert channel to be effective, it needs a bandwidth of greater than 100 bps [29]
3. The channel supports either sending one bit at a time or multiple bytes at a time
4. The channel is repeatedly used by the sender and receiver to transmit data.
5. When an application writes a value to the channel, it overrides the previously written value. Only one value is valid at any point of time.

Table 2.1: Overt and Covert channels in Android

Channel Name	**Type**	**Level**	**Synch**	**Remarks**
Shared Pref	Overt	Application	Async	
External/Internal storage (files)	Overt	Application	Async	Reading/Writing to files for communication
Intents	Overt	Application	Async	
System Log	Overt	Application	Sync	Reading/Writing to logs for communication
UNIX Socket Communication	Overt	OS	Sync	Reading/Writing to sockets for communication
Bluetooth/NFC	Overt	Application /OS	Sync	Reading/Writing via BlueTooth/NFC for communication
System Settings	Covert	Application	Sync	each setting can have only one valid value at a time. writing a new value overwrites the older value
Automatic Intents	Covert	Application	Sync	Limited number of intents exist that support broadcast operations
Threads Enumeration	Covert	OS	Sync	Uses the /proc, /sys for operation
UNIX SocketDiscovery	Covert	OS	Sync	Communication viachecking if socket isopen/closed - Open is bit 1, Closed is 0.
Free space in FS	Covert	OS	Sync	encode data as a seriesof free space values. Fillup the disk accordingly.
Reading /proc, /sys	Covert	OS	Sync	Uses the /proc, /sys for operation
Battery	Covert	OS	Sync	same as automatic intents
Phone	Covert	OS	Sync /Async	Use the last called number (ASCII text) for communication. API supports only last called number

3. SURVEY

3.1 Overview Of Security Challenges - A Classification

Android ecosystem has several security challenges despite being designed with security in mind. We classify challenges based on the components they manifest in. This helps us understand security challenges in Android ecosystem easily. An overview of our classification is found in Fig. 3.1.

3.1.1 App Marketplaces

The major challenges that marketplaces face are summarized below:

- Efficient and automatic app vetting process (for malware and malware-like behavior)
- Integrity of developer identity and safe delivery of content
- Fragmented and numerous marketplaces

3.1.1.1 Efficient And Automatic App Vetting Process

There were incidents where malware were distributed through Google Play Store and other marketplaces. Malware authors are not deterred from releasing their malware through Google Play Store as the one-time cost to start publishing an app is very low ($25) compared to the profit from their malware. Changing the fees structure in an attempt to curb malware from being published, would deter some legitimate app developers from using Play Store.

It can be seen that malware poses a serious threat to marketplaces. Stopping it from spreading keeps the marketplaces clean. Cleaner marketplaces attract more

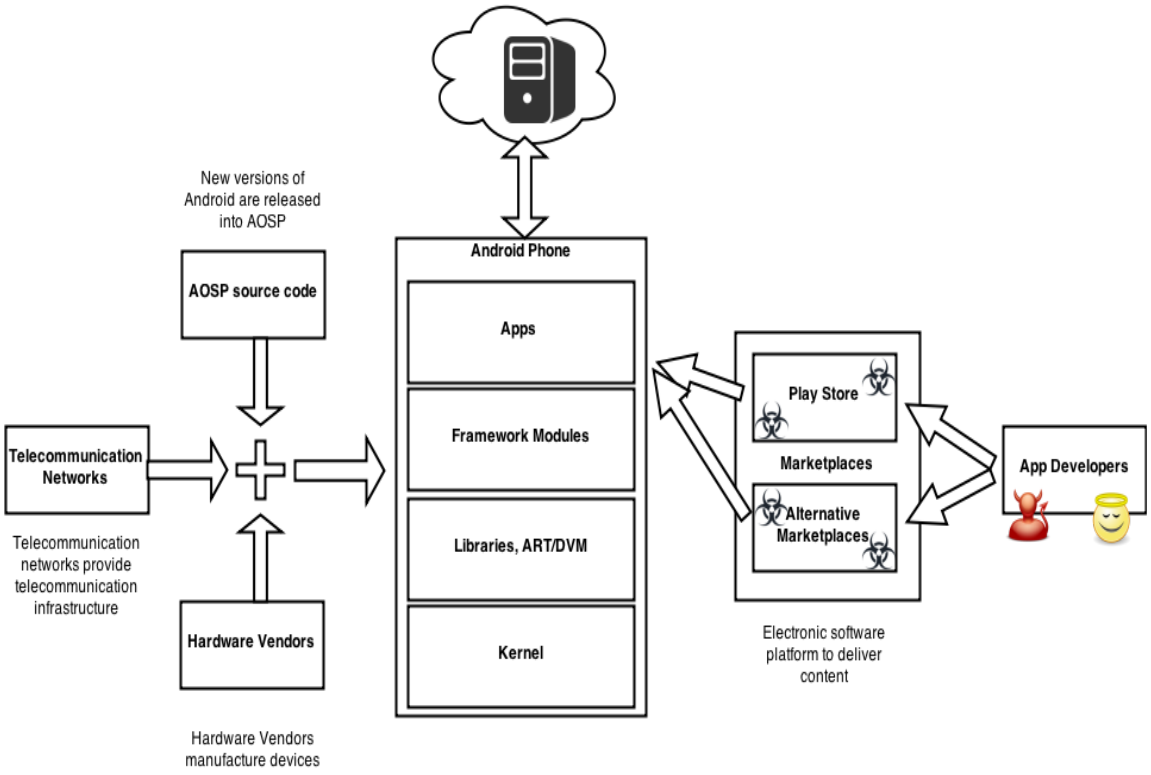

Figure 3.1: Overview of Android Survey Taxonomy showing components of Android ecosystem. Components include Google's AOSP, hardware vendors, telecommunication (carrier) networks, the Android Phone (refer Fig. 3.2 for more details), app developers (malicious and benign) and several App Markets (malicious and benign). A phone is complete using AOSP source code from Google, hardware from hardware vendors and telecommunication infrastructure from carrier networks. Users may download additional software from App Markets. App developers may release their apps to several marketplaces.

users (as users prefer cleaner marketplaces to those infested with malware). A marketplace frequented by many users attracts more developers (to release apps). A wide range of developers and apps enriches a marketplace. A vibrant marketplace attracts more users and the cycle of market-users-developers repeats itself thereby increasing the hygiene of the ecosystem.

It is evident that clean marketplaces increase the hygiene of the ecosystem. Hence,

all marketplaces must put in place vetting measures to address the threat of malware. The ideal vetting process will be highly efficient and fully automatic removing all malware apps and apps exhibiting malware like behavior from marketplaces. As several thousand apps gets submitted per day, human involvement in the vetting process should be as minimum as possible. Also, the process should be efficient to scale well. To this end, Google runs a security service called *Bouncer* [30] which is believed to check apps for known malware and malware like behavior. Security measures in alternative marketplaces are largely nonexistent.

3.1.1.2 Integrity Of Developer Identity & Safe Delivery Of Content

Earlier we saw that around 1 million apps exist in the PlayStore alone. The total number of unique apps may increase when we consider all marketplaces. This deluge of apps can be largely attributed to the popularity of Android powered devices and also to the ease of publishing apps to marketplaces. While Google Play Store requires a developer to pay a one-time nominal amount to start publishing apps, alternative marketplaces merely require the publisher to have a valid and active email address to publish their apps. This ease of publishing comes at a serious price.

It is trivial to create an email account using fake details. Once a fake email account impersonating a trusted developer is created, it is easy to trick people to install apps from the fake developer. This may also lead the users to install malicious apps (*malware*). Thus, marketplaces must take more efforts in establishing non-repudiable identity of developers. Once a developer identity is established, it can be used in enforcing security policies. In addition to establishing identity, marketplaces must provide secure downloads to prevent Man-In-The-Middle attacks.

3.1.1.3 Fragmented And Numerous Marketplaces

In section 2.1, we presented a (non-exhaustive) list of several marketplaces that developers may release apps to. There are many more marketplaces that serve specific niches of Android ecosystem. This proliferation and specialization of app marketplaces gives Android users a wide range of options to choose from. However, having multiple marketplaces becomes a major challenge for enforcing security policies. If there were limited marketplaces, security policies may be enforced easily (though that might limit the choices users may have). While a consolidation of marketplaces may seem impractical, all marketplaces must put in measures to ensure the integrity of developer identity, safe delivery of content and removal of malware from marketplaces.

There has been significant work in the literature that identifies security shortcomings of app marketplaces and proposes enhancements that can be integrated to existing app markets.

3.1.2 Hardware Vendors

Hardware vendors manufacture devices by assembling hardware, installing OS and other software. Security challenges in this component are listed below

- Risk due to fragmented OS distribution and slow (or no) updates
- Risk due to Vendor specific software
- Risk due to Vendor specific hardware

3.1.2.1 Risk Due To Fragmented OS Distribution And Slow (Or No) Updates

Android ecosystem suffers from a critical security challenge due to hardware vendors' phone support policy.

Hardware vendors discontinue support for phones after a few (about 2) years leaving users with out-of-date and heavily vulnerable software. To understand the gravity of this situation, we find that as of Dec 1, 2014 there are about 10% of Android phones that still run GingerBread OS [31]. A quick search in the CVE database for Android reveals that GingerBread suffers from several high impact security issues.

While it may be reasoned that users may install the latest OS version (if available) from external sources, it must be taken into account that not all users will be technical enough to install (flash) a custom ROM onto their phones. Additionally, there is the risk of bricking the phone. Moreover rooting a device may not possible due to legal restrictions. These technical/legal difficulties aggravate the problem - first the software is vulnerable and second the software might not be easily replaced (requires rooting a phone and/or flashing a custom ROM).

This is a crucial security challenge as users are forced to use the vulnerable software until they replace their devices. We also note that this is an area that needs a lot of work to address this seemingly trivial yet critical challenge.

*3.1.2.2 Risk Due To Vendor Specific *ware*

It is a common practice for hardware vendors to provide extraneous components (such as apps, widgets, drivers) that come pre-installed with every phone they release. As was noted earlier, hardware vendors are allowed to use permissions that fall into the highly privileged *Signature* and *SignatureorSystem* categories. It is imperative that components using these privileges must be free of any vulnerabilities.

Thus it can be seen that hardware vendors have a profound impact in the security of Android ecosystem. Yet this is one of the least addressed areas in security research of Android ecosystem.

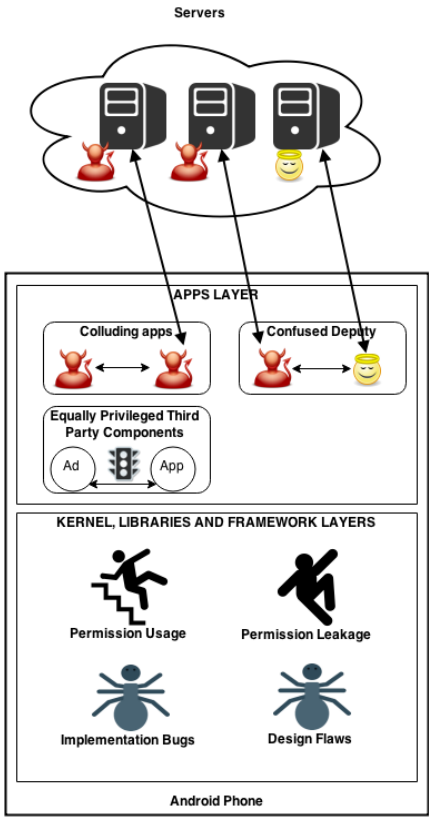

Figure 3.2: Security issues in Android phone - Overview. Security issues include but not limited to abuse of hardware by malware, vulnerable libraries, confused deputy attacks, privilege escalation attacks, data leakage attacks, vulnerable apps, malware apps/libraries, co-existence of ad libraries and apps in the same process space.

3.1.3 Linux Kernel, Libraries And Framework Layer

The Linux kernel serves as an interface between the higher level framework modules/apps and the lower level hardware devices such as the display unit, camera, audio, Wi-Fi and internal flash memory devices. The framework and libraries layer along with the Dalvik run-time provide the greatest portion of Android's functionality. Readers are referred to sections 2.1.4.1 through 2.1.4.4 for background information on these layers.

Though the kernel, libraries and framework layers are often depicted with well

defined boundaries, all three layers are essentially libraries that are used by the upper layers of Android. Considering them this way helps us bring out their similarities and consolidate security challenges in these layers. An illustration of the security issues in Android phones is shown in Fig. 3.2.

The security challenges in these layers are listed below:

- Implementation Flaws in the kernel, libraries and framework layers
- Design Flaws in the kernel, libraries and framework layers
  - Design flaws that lead to abuse of kernel artifacts
  - Design flaws that lead to abuse of hardware
  - Design flaws that lead to several permission usage/leakage issues
  - Design limitations due to Discretionary Access Control

3.1.3.1 Implementation Flaws In The Kernel, Libraries And Framework Layers

Security issues due to implementation flaws are primarily due to bugs in implementation. For example, incomplete validation of function parameters in a function of futex.c allowed local users to gain root privilege or allowed them to crash the kernel. This is a serious security issue that originated due to a bug in implementation.

3.1.3.2 Design Flaws In The Kernel, Libraries And Framework Layers

Some security issues arise due to flaws in designing the usage of modules on Android. For example, a common attack vector in Android phones is the abuse of /procfs and /sysfs. The functionality of these components is very similar to that of Linux's /procfs and /sysfs. While the use of /procfs and /sysfs by themselves is not a security issue, the use of such components in the Android kernel without

requiring permissions is a security issue. Several work have utilized these design flaws to extract sensitive information from apps. We present such work in section 3.3.3.1.

Common security challenges due to design flaws are

1. flaws that lead to abuse of kernel artifacts
2. flaws that lead to abuse of hardware
3. flaws that lead to abuse/bypass permission model such as over-privileged apps, native code permission issues and permission leakage

Permissions are fundamental to app isolation in Android. Permissions are enforced in two layers - in framework layer and in kernel layer [32]. Kernel enforces permission checking when the app requests network/file-system operations. The framework layer enforces permissions that were allowed by the user during installation.

One challenge is the effectiveness of permission request screen. The effect of app permission request screen is studied in the literature. Researchers have found the permission warning screen to be largely ineffective as there is little context to understand why an app requests permissions and how the permissions are used. Researchers have also proposed new ways of making it effective. These works are presented in section 3.3.3.3.2.

Another challenge is over-privileged apps. Over-privileging an app occurs when an app requests for permissions beyond those that it actually requires. Over-privileged apps are a security risk as they request more information/access than they need (violates principle of least privilege) and also increases the attack space. Section 3.3.3.3.2 presents solutions that address the problem of over-privileged apps.

Apps may use custom/third party native code (C/C++/assembly) libraries for high performance components. These libraries are given the same privileges as that

of the app using them. This is a security risk since untrusted third party code is given the same privilege as that of trusted code. Section 3.3.3.4 presents solutions to this challenge.

Yet another challenge is the risk of privilege escalation/leakage. Privilege escalation/leakage could allow a low permission app to gain permissions of a high permission app. We present solutions that address this challenge in section 3.3.3.4.1.

Orthogonal to the above flaws, we look at a design limitation of Android's access control mechanism. Android OS runs on a modified Linux kernel which employs Discretionary Access Control (DAC) as the resource access control mechanism. DAC restricts access to objects based on identity of entities they belong to. Users can use their discretion to share (with others) access to resources they own. Complementary to DAC, another access control mechanism called Mandatory Access Control (MAC) exists. MAC allows centralized policy enforcement of access control and provides system wide guarantees for controlling access. Individual users cannot modify these policies either intentionally or accidentally. MAC can protect resources better and makes enforcing security policies easier than DAC. We look at solutions that advocate the use of MAC in the kernel layer to increase system security in section 3.3.4.

3.1.4 Android Apps

Apps give the Android ecosystem its extensibility and are thus a critical part of the ecosystem. Security issues in this layer can be broadly classified into two

- Security risks due to incorrectly used API issues (Benign apps)
  - Issues with SSL APIs
  - Issues with equally privileged Third Party Components
- Security risks due to malicious Behavior of apps (Malicious apps)

3.1.4.1 Risks Due To Incorrectly Used API Issues (Benign Apps)

Android's libraries layer, framework layer and third party libraries provide API for apps to use. Some of these API may be simple to use while others are complicated. When developers use these API in their apps, they may *inadvertently* use them incompletely/incorrectly. Such incomplete/incorrect usage could be susceptible to vulnerabilities. A befitting example is the incomplete implementation of SSL in apps due to which attackers can decrypt encrypted content on the fly.

It is common practice for a free app developer to make money from his/her app by adding an ad library. The ad library serves ads to users and the developer receives money whenever a user clicks on an ad. The ad library lives in the same process space as that of the host app. Thus, the app can interfere with the ad library's functionality and vice-a-versa. This puts both the app and the ad library at risk. Every developer would want his app's data to remain secure. Similarly an ad library provider would want to prevent bogus clicks on ads. The app and the ad library may be benign by themselves but they need to protect themselves from being abused due to incorrect usage of their API.

In section 3.2.1.1 we discuss these challenges and their solutions in detail.

3.1.4.2 Risks Due To Malicious Behavior Of Apps (Malicious Apps)

The bigger challenge in the app layer is the prevalence of malware. Malware are released *intentionally* to exploit vulnerability in apps/OS. Malware could be released as an app/as a library in other apps.

Android OS was designed to ensure isolation of apps. All apps request for permissions at install-time. Users may choose to grant/deny all permissions to proceed with/abort installation. This policy works good for securing apps in isolation but it does not prevent two or more seemingly innocuous apps from colluding with each

other to expose user's private sensitive info. We present several solutions that address *colluding apps* in section 3.2.2.

Alternatively, a malicious app may exploit a benign app to achieve a malicious end. This is called as the *confused deputy* attack. The confused deputy is a benign app (the victim app) that receives a (malicious) command from a malicious app. It assumes the command to be legitimate and faithfully executes it thus helping a malicious app in infecting/extracting sensitive info.

Privilege escalation occurs when a malicious app, with little or no permissions, requests a legitimate app with desired sensitive permissions, to execute privileged tasks on its behalf.

Repackaged apps affect developers adversely as cloned apps try to cash in on the victim app's popularity. Clone-ware also affects users as they may be used as vectors for malware distribution. However, given two apps that are closely similar to each other, it is not possible to label one app the clone of the other without sufficient ground truth. Due to this limitation, a common solution is to identify pairs (or groups) of apps that are clones of each other and provide these pairs (or groups) for manual analysis and resolution.

We present several categories of solutions that address this challenge. Section 3.2.2.1 presents isolation based approaches. Section 3.2.2.2 present approaches that trace system calls and monitor for malicious calls. Section 3.2.2.3 provides solutions that use tainting to address the problem of malware. Finally, we present signature based or policy based solutions in section 3.2.2.4.

3.1.5 Defense In Depth

It may be recollected that we listed malware as a security challenge in the marketplaces and also in the apps layer. Similarly, we listed confused deputy attacks as

a security challenge in kernel layer and the app layer. These issues are too broad to be solved completely at any one component of the ecosystem. For example, malware vetting processes in the marketplaces have to be very fast, efficient and fully automatic. Malware removal mechanisms at the app layer may use well-defined rules and detailed monitoring over a relatively longer duration of time. Having multiple layers of defenses provides sufficient redundancy in addressing the ever-growing challenge of malware.

This is a practical example of defense in depth. Each component of the ecosystem can address a challenge as best as it can - and multiple layers come together to address a challenge comprehensively.

3.2 Android Apps - Security Challenges And Their Solutions

It may be recollected from section 3.1.4 that the challenges in the Apps layer are:

- Risk due to incorrectly used API issues (Benign apps)
  - Incomplete implementation of SSL in apps
  - Incorrect use of Third Party Components
- Risk due to malicious behavior of apps (Malicious apps)

Identifying and solving security issues in applications layer (Fig. 2.1) has received a lot of focus in the literature. A systemization of solutions to address the above challenges is shown in Fig. 3.3.

3.2.1 Risk Due To Incorrectly Used API Issues And Its Solutions

The solutions to address the challenge of incorrectly used API may be classified according to the problem they solve:

- Solutions to incomplete use of SSL in apps.

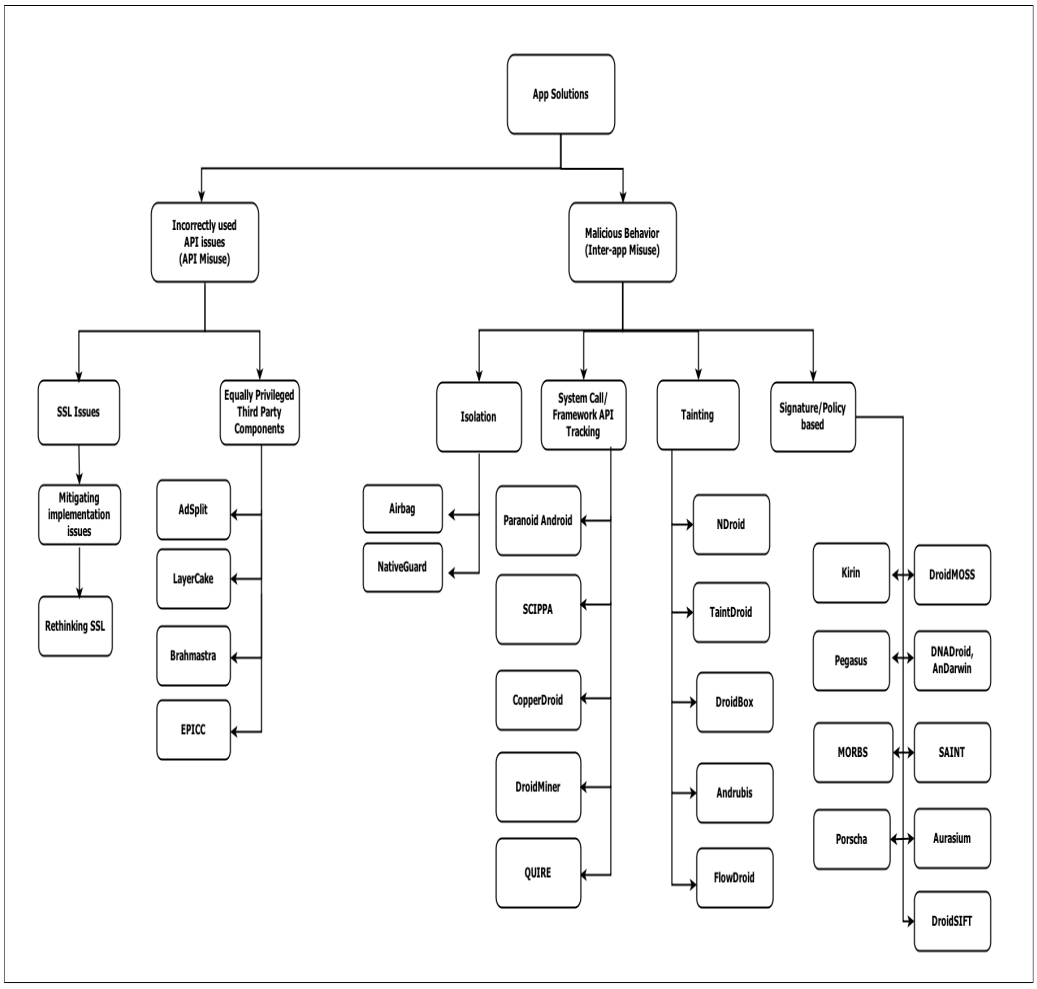

Figure 3.3: Solutions to security challenges in the App layer

- Solutions to incorrect use of third party components

3.2.1.1 Incomplete Implementation Of SSL In Apps

Security bugs in apps using SSL primarily arise due to an inadequate understanding of the working of SSL. Moreover, Android expects developers to implement code for verifying digital certificates. Not all developers are fully aware of the inner workings of certificates to correctly implement the required code. This makes matters worse when developers may not understand the consequence of the choices. This

inadvertently opens their application to malicious attacks.

To demonstrate the same, Sounthiraraj et al. [33] built an automated detection mechanism called *SMV-Hunter* to identify man in the middle vulnerabilities (MITM) in apps that use SSl. To identify such vulnerabilities in an app, they used a combination of static analysis and dynamic analysis. SMV-Hunter used static analysis to build a method call graph and to identify the entry points of the app. Using a custom UI automation tool, SMV-Hunter runs dynamic analysis to identify MITM vulnerabilities. To mitigate the search space explosion problem, static analysis guides their dynamic analysis module.

Orthogonally, Fahl et al. [34], [35] conducted vulnerability analysis on SSL implementations in apps. They used static and manual analysis to identify common SSL implementation issues - such as developers

1. trusting all certificates (without verifying),
2. using self-signed testing certificates in production builds,
3. passing sensitive data in plain-text and
4. not guarding (their apps) against SSL stripping [36].

These issues can be exploited to expose encrypted data in transit. Due to the complexity of SSL implementation, developers "get-it-to-work" code freely available from blogs and programming assistance websites. While these codes work, they may not always ensure that the data being sent via SSL is secure and immune from being eavesdropped.

To address this problem, it can be observed that it would benefit developers to merely configure SSL options rather than implementing SSL functionality. So, they

built a framework module that provides all required SSL functionality and overrides existing SSL implementations. Developers can configure the module's settings according to their needs but cannot disable/override the security options.

3.2.1.2 Incorrect Use Of Third Party Components

It is common knowledge that free apps embed ad libraries (third party libraries)to drive revenue. However, these libraries run at the same privilege level as that of the apps. Thus, both the app and ad library can be manipulated by each other.

Common attacks that are possible are

1. display forgery,
2. size manipulation,
3. input forgery,
4. click-jacking,
5. focus stealing,
6. ancestor redirecting,
7. data privacy attacks (eavesdropping attacks).

Shekhar et al. [37] proposed mitigating these attacks by splitting the ad library and the app into separate processes with different UID (*AdSplit*). Since Android implements uid based isolation (inherent to Linux), the two entities will be isolated from each other thus preventing the above mentioned attacks. Once split, the manifest file (AndroidManifest.xml) is updated to reflect the splitting and the apk file is regenerated.

A major drawback with AdSplit is that the app's signature is broken when it is disassembled and packaged back. This can be avoided if Android provides a mechanism to embed third party components into the app's UI. Roesner et al. [38] designed and built *LayerCake*. LayerCake adds a new activity called *EmbeddedActivityView*. Using the EmbeddedActivityView, it is possible for developers to design user interfaces with components embedded in them, control access to them without worrying about the aforementioned security issues.

It may be seen that these solutions specifically target the ad libraries and other third party components that need to be displayed on the UI. However, several third party components - such as native code libraries - are used by apps. These components also run at the same privilege level as the app and are still vulnerable to incorrect usage. Section 3.2.2.1 presents solution to this problem.

Bhoraskar proposed *Brahmastra* [39] to help test the third party components inside an app. Due to the asynchronous nature of Android, it is quite possible for the search space to explode when running dynamic analysis. To help limit the search space, the authors performed static analysis of the app and identified execution paths that test a third party library inside an app. Once the paths are identified, Brahmastra rewrites the app to automatically test these execution paths. To reduce the time taken to test, Brahmastra jump starts from a random node on the execution path optimistically. If it fails to execute properly it picks an earlier node to start from. Since it targets the paths to reach the third party component, it does not suffer from the space explosion problems traditional UI testing tools face. This makes this technique a good option to test third party components for vulnerabilities/threats.

Octeau et al. developed a tool *EPICC* [40] to help analysts understand the flow of data/control across apps. The tool mapped all inter component communication by redefine Inter-Component Communication (ICC) as an Interprocedural Distributive

Environment (IDE) problem. IDE has been well studied to identify data and control flow between multiple procedures. The authors identified all entry and exit points and matched each exit point to an entry point. However, it should be noted that this work merely identifies inter-component communication and should be used along with several other tools to understand malicious app behavior.

3.2.2 Risks Due To Malicious Behavior Of Apps And Their Solutions

Several solutions have been proposed to address the challenge of malicious apps. They may be broadly classified into the following:

- Isolation based approaches
- System call/Framework API tracking approaches
- Tainting based approaches
- Signature/Policy based approaches

We describe each of the approaches in detail below.

3.2.2.1 Isolation Based Approaches

Solutions in this category employ some form of isolation to separate malicious entities (apps/components) from benign ones. Isolation can be achieved by the use of virtualization or by the use of process isolation. Major challenges for solutions in this category are

1. Overhead incurred due to isolation mechanisms
2. Effectiveness of isolation
3. Selective behavior of malware (VM-aware threats)

Wu et. al. proposed *Airbag* [41], a virtualization based isolation approach to isolate known good applications from unknown applications. To reduce the overhead of virtualization, Airbag shares kernel space among multiple virtual instances. Each instance has its own copy of

1. App Isolation Runtime (AIR),
2. filesystem and namespace instances,
3. services and daemons.

Airbag also provides mechanisms to multiplex accesses from native and all Airbag instances. All trusted applications run on one instance (the native instance). Each untrusted application runs in a separate Airbag instance. Thus the unknown app is isolated from every other instance and thus cannot affect any benign apps/services.

However, the user needs to specify which app needs to be isolated. It should be noted that Airbag is that the basic unit of isolation is an app. If a benign app has a malicious native library, then we have to isolate the entire app to isolate and contain untrusted behavior.

Sun et. al. proposed *NativeGuard* [42] to handle such malicious native libraries. Native libraries are commonly used by apps for performance benefits (compared to Java code). NativeGuard protects benign apps from malicious native libraries by

1. Isolating native libraries into a separate process - thus they cannot access app's resources
2. Limiting permissions - not providing the native component all permissions that the app has

To support JNI calls in this separated environment, a stub and a set of trampolines are set up which help in executing JNI calls. It may be recalled that this

approach is similar to AdSplit [37] except that NativeGuard achieves isolation of native libraries whereas Adsplit achieves isolation of ad libraries.

3.2.2.2 System Call/Framework API Tracking Approaches

System call tracking is a common technique used in addressing the threat of malware. The challenge in solutions is the limited resources available on mobile devices. Smart-phones have limited computation power and are limited in the support of computationally intensive tasks.

Portokalidis et al. [43] proposed *Paranoid Android*. Paranoid Android records system calls and sends them to a remote server. The remote server executes the same system calls on an emulator fitted with several security mechanisms. If any of the security mechanisms flags the call as malicious, then the remote server can remotely kill execution on the phone. Paranoid Android suffers from practical complexities due to asynchronous calling mechanism, IPC, concurrency issues and manipulation of stored execution flow. Moreover emulator aware malware may not exhibit their malicious behavior when run in an emulator.

Reina et. al built a behavior analysis tool that aides in malware analysis. *CopperDroid* [44] is a QEMU based out-of-box malware modelling tool. It tracks all system calls by monitoring the change in processor privilege level from supervisor (privileged) mode to normal (user) mode. For each system call, its parameters are inspected. Since there are innumerous system calls, the authors build a *unmarshalling oracle* which can unmarshall all parameters of a system call. The tool outputs the system calls and re-interprets the system parameters in a human-readable format for quicker understanding of how malware behaves.

Yang et al. proposed *DroidMiner* [45], an in-box malware modeling tool that tracks framework level APIs, Content Providers. DroidMiner employs machine learn-

ing extensively to model malware. For each app, a signature based on the framework APIs used is generated and encoded as a bit vector. Association rule mining algorithms are run on this bit vector to identify the most common API that malware uses. When a new app is encountered, a signature is generated for it and it is checked against the rules generated. Based on the output of the checks, DroidMiner decides if an app is malicious or not.

Dietz et al. [46] addressed the problem of privilege escalation by tracking provenance. Provenance identifies origin from which information flows. An IPC request is added to a call stack showing all the upstream callers. If all the callers have been granted the desired permission, the IPC request is allowed to go through. Otherwise, it is blocked immediately.

Backes et al. [47] proposed *SCIPPA* on the similar lines of QUIRE [46]. The primary difference between SCIPPA and QUIRE is that QUIRE expects the app to generate the IPC call stack whereas in SCIPPA the system builds the call stack. Once such a stack is constructed, the call stack is sent to the recipient process for permission checks. The recipient process may check if all applications in the call stack have the required permissions to execute an operation or may abort the call if not. Both SCIPPA and QUIRE use provenance of system calls as a mechanism to thwart privilege escalation, confused deputy attacks, intent spoofing and intent hijacking. Though privilege escalation and confused deputy attacks are addressed, data leaks due to colluding apps are not addressed by these mechanisms.

3.2.2.3 Tainting Based Approaches

Taint analysis tracks the flow of externally modified input and warns if the flow reaches sensitive system functions. Tainting is a form of black-listing.

Enck et al propose a taint-based mechanism to identify and prevent sensitive

information from leaking from the phone. Their tool *TaintDroid* [48] taints and tracks sensitive information flow between two apps or between the system-app to automatically identify information leak. TaintDroid modifies the framework layer and the Binder layer heavily. However, TaintDroid is limited to Java based code and does not track sensitive information leak via the native code. Subsequently a tool called *DroidBox* based on TaintDroid was developed and made available for public use [49]. Yet another tool named *Andrubis* was developed on top of TaintDroid and DroidBox. Andrubis is offered as a on-demand malware detection service [50].

While TaintDroid (and other tools built on top of it) provide tainting of Java based code, native code tainting was largely left unexplored. Qian et. al. proposed *NDroid* to taint and track flow of sensitive information across native code. NDroid is implemented on the QEMU emulator with the following changes.

1. a native instruction tracer
2. a DVM hook engine to track JNI related functions
3. a system library hook to track system calls and
4. a tainting engine to propogate taint across the native layer.

Arzt et al. proposed *FlowDroid* [51] a tainting mechanism that primarily uses static analysis for taint propogation and for identifying information leak. In sharp contrast with TaintDroid and other tainting mechanisms which employ dynamic analysis, FlowDroid is a static analysis mechanism. Arzt et al. propose mechanisms to model an app's lifecycle by statically analyzing AndroidManifest.xml and the app's code. Once modeled, taint tracking is made available by forward taint propogation and an optional on-demand backward analysis to reduce the false positives that may be incurred due to static code analysis. It is notable that FlowDroid does not suffer

from problems such as malware detecting the presence of detection mechanism (such as TaintDroid) or code coverage. Yet FlowDroid would fail considerably when run on obfuscated code or encrypted code.

3.2.2.4 Signature/Policy Based Approaches

Solutions under this category use signature/policies to detect and stop the spread of malware. These solutions can be implemented as framework level or app level.

3.2.2.4.1 Framework Level Solutions

Enck et al. [52] proposed *Kirin* - a lightweight app certification procedure. Kirin define rules that indicate suspicious behavior. Once identified, the rules are stored on the phone. During an app's installation, Kirin validates app's permissions against the rules previously stored. If a rule matches, installation of the app is blocked. If not, installation proceeds to completion. While this model works well for known patterns, it falls short when an unknown pattern is encountered. Hence this solution can only be used in conjunction with other security solutions.

Chen et al. [53] designed *Pegasus* based on the assumption that malicious apps use API differently than benign apps. The idea is to use the permission event graphs (PEG) to identify context and sequence of API used. Using static analysis, PEG captures the application artifacts (buttons etc), system artifacts (events etc) and the interaction between the two. From the constructed PEG, event sequences are extracted and policies are defined. A run-time monitor enforce the policies on the device. They aim this tool to be one of the many tools that analysts use to identify malware.

Wang et al. [54] studied the threat of unauthorized origin crossing to address the threat of confused deputy attacks (permission leakage/permission escalation). They

propose *MORBS* to label every message with its origin. Developers specify white-lists policies to allow certain origins and black-lists to deny certain other origins. A reference monitor is employed to enforce the white-lists/black-lists policies during run-time thus preventing unauthorized origin crossing attacks. The authors also suggest building origin based protection mechanism into the Android platform.

Zhang et al [55] proposed *DroidSIFT* a partial classifier as against the signature based or machine learning based binary classifiers. They built a database of dependency graphs for apps containing contextual information such as entry points of apps, constants in the API. The weights in the graph are assigned automatically with heavy weights assigned to sensitive API and lighter weights assigned to regular API. Based on these graphs, bit vectors are generated for benign and malware apps. For a new app, they extracted features and check with the features generated from the database. Based on the percentage of match, they claim that the app is malware/benign.

3.2.2.4.2 App Level Solutions

Previously we presented that operated on the system level. We now look at solutions that operate in the app layer. Ongtang et al. [56] identify the need to move toward app-centric security model from the current system security model. The authors suggest that apps should be able to specify conditions when other apps can interact with it. They should be able to provide a white-list/blacklist of apps they can interact with and they should also be able to exercise security requirements. These protections should be available in addition to the system protections. The authors design *SAINT* with install-time and run-time enhancements to Android.

Xu et al. [57] proposed an app level mechanism called *Aurasium* to mitigate privacy losses due to colluding apps. Aurasium repackages application to harden

them. Since the original signature is lost, the authors propose using unique certificate for each unique vendor. This proposal allows the use of shared userid. The authors define policies for ensuring privacy, preventing SMS, MMS abuse and preventing privilege escalation. The enforcement of the policies is done by the Aurasium Security Manager which serves as a monitoring mechanism.

Contrary to above approaches, Ongtang et al. [58] addressed the issue of policing content on the phone to solve the following Digital Rights Management (DRM) issues - binding content to phone, allowing content to be accessible by specific apps and facilitating additional constraints on the usage of content.

During transmission of content, the authors protect content in transmission by using Identity Based Encryption (IBE). While the content resides on the phone, a reference monitor is used to mediate and control the policy.

Solutions that address the issue of repackaged apps use the intuition that repackaged apps will be very similar except for a few changes. A challenge for these solutions is the presence of library code.

- *DroidMOSS* [59],
- *DNADroid* [60] &
- *AnDarwin* [61] that address the issue of cloneware.

DroidMOSS [59] tackled the issue of repackaged apps (cloneware) by the use of Fuzzy Hashing to identify similarities between apps.

1. Feature Extraction: This involves statically extracting DEX bytecode from the apk file.
2. Fuzzy hashing: Once dex code is extracted, the authors use a sliding window algorithm to break the entire source code into smaller pieces. Each of these

pieces is hashed individually. Finally all the intermediate hashes are hashed again to generate the final hash for an app.

3. Similarity Matching: Once the hashes are generated, DroidMOSS uses a sequence matching algorithm to identify the edit distance and thus the similarity between two applications. Repackaged applications will be very similar except for a few changes. To prevent noise due to ad libraries, the authors employ a white-list based approach to discard library code.

Crussell et al. [60] proposed *DNADroid* to identify cloned apps based on similarity in description and metadata. Using description and metadata similarity is a fairly reasonable assumption since cloneware seek to mislead users by posing as official apps. Once a pair of apps is identified, they built a program dependency graph for each app. The two graphs are compared to identify pair-wise similarity. The similarity score for a pair of cloned apps will be significantly high. DNADroid outputs the pair of apps along with a similarity score.

While this approach is reasonable, identifying possible pairs for an app is a time-consuming task given the abundance of apps in the marketplaces. Crussell et al. [61] proposed *AnDarwin* To address this limitation.

For each app (similar to DNADroid) program dependency graphs are built. These graphs are used to represent the app by vector notation. Since all apps are represented by vectors, identifying similarity is reduced to the problem of identifying similarity of vectors. Location Sensitive Hashing (LSH) is used to cluster similar apps. Jaccard's index is used to measure similarity. To prevent identifying these libraries as false positives, AnDarwin excludes such library code from similarity scores using a threshold based approach.

3.3 Kernel, Libraries And Framework Layers - Security Challenges And Their Solutions

In this section, we discuss the security challenges in the kernel and framework layers and their solutions in the literature. The challenges in this layer can be categorized as

- Security risks due to Implementation Flaws in the kernel, libraries and frameworks layers
- Security risks due to Design Flaws in the kernel, libraries and frameworks layers
  - Design flaws that lead to abuse of kernel artifacts
  - Design flaws that lead to abuse of hardware
  - Design flaws that lead to several permission usage/leakage issues

An overview of all security issues in these layers is available in Fig. 3.4

3.3.1 Kernel - Security Issues Due To Implementation Flaws And Their Solutions

The kernel is responsible for interfacing with the hardware and for providing features such as

1. memory management,
2. process management,
3. file system management,
4. inter-process communication etc...

The kernel runs in supervisor mode which is highly privileged when compared to user mode. Hence, any implementation flaw in this layer would prove very detrimental to security of user data on the phone.

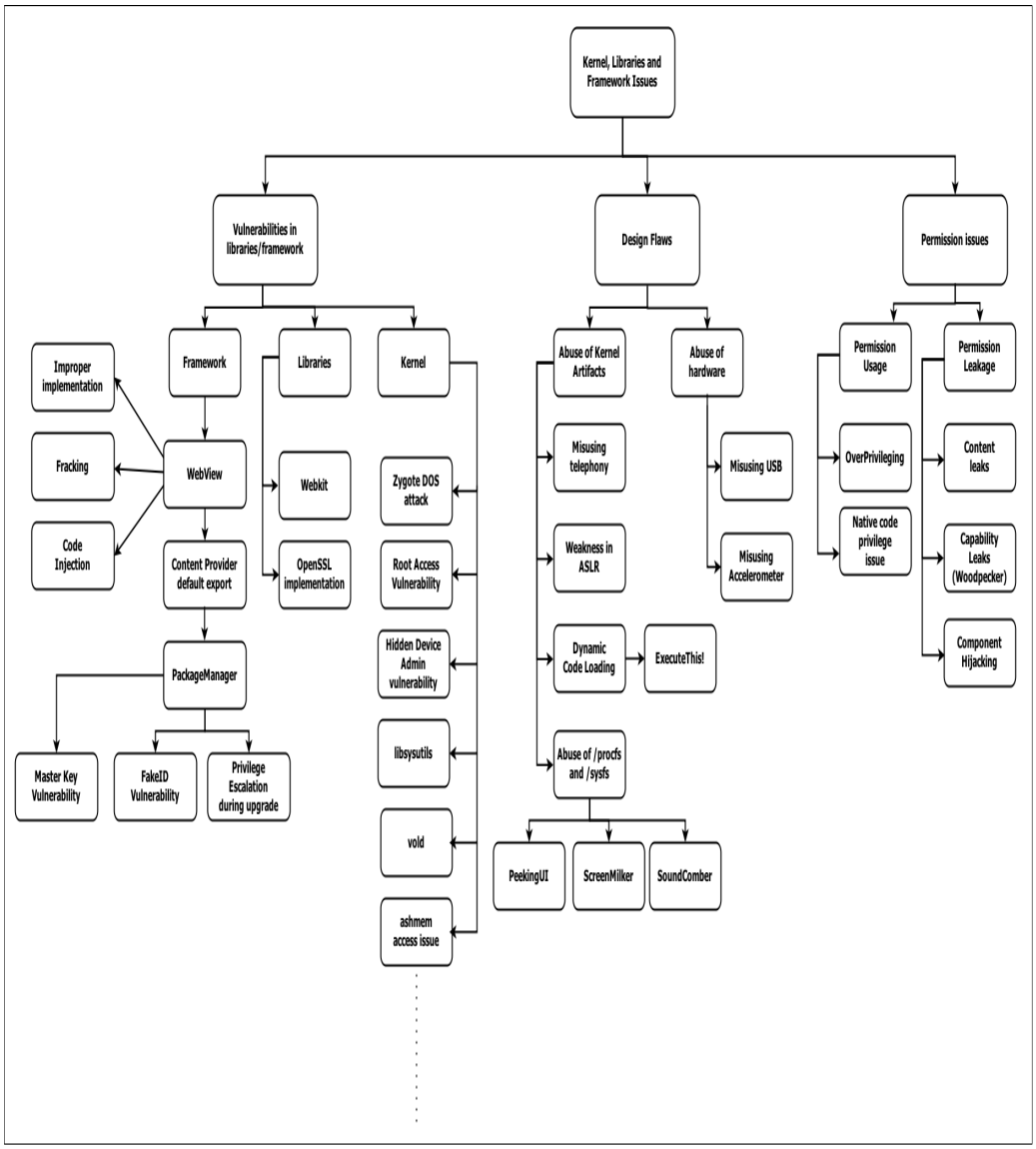

Figure 3.4: Security challenges in the Kernel layer

Several examples that show the gravity of exploitable implementation flaws are as follows:

1. Zygote DOS attack

2. Root Access Vulnerability

3. Hidden Device Admin vulnerability

4. libsysutils

5. vold

6. ashmem access issue

3.3.1.1 Zygote DOS Attack

Zygote is a daemon that is launched during system boot process, enables code sharing across VM instances and is responsible for launching apps in Android. Zygote listens on a socket to receive requests from the SystemService component to launch processes. The SystemService component is not owned by the root whereas Zygote is owned by the root. To allow the non-root process to communicate with the root process, the permissions on Zygote were set to be world readable and world writable (666). This allowed arbitrary apps to send requests to Zygote which were executed. When the socket is flooded with requests, it creates processes for each request and quickly runs out of memory. This triggers safety mechanisms in Android which reboot the device. Thus, the device can be forced into an endless reboot loop. More details about the attack can be found at [62].

3.3.1.2 Root Access Vulnerability

[63] discusses about a root access vulnerability that can be exploited to achieve supervisor privilege or crash the kernel. Using a specially crafted FUTEX_REQUEUE command, it is possible for local users to gain root privilege since the futex_requeue function of kernel/futex.c does not check that the two futex addresses provided are different. Once supervisor access is achieved, the local user may crash the kernel or to execute arbitrary commands under the high privilege mode.

3.3.1.3 Hidden Device Admin Vulnerability

[64], [65] present a malware that achieves root privilege and installs itself as a device administrator. The malware uses several well known exploits to gain root privilege. After installation, the malware hides itself so that the user does not see the device admin. Once device admin privilege is gained, the malware can irreversibly modify the contents of the phone. It is also possible for the malware to reset the phone to factory state.

3.3.1.4 Libsysutils

Android versions 2.2.x through 2.2.2 and 2.3.x suffer from a buffer overflow in libsysutils shared library. By sending the wrong number of arguments to FrameworkListener::dispatchCommand method, a use-after-free error is triggered [66]. This exploit allows remote attacks to execute arbitrary code.

3.3.1.5 Vold

The volume manager daemon (vold) allowed local users to gain root privileges [67]. By sending a negative index to a incompletely validated signed integer check in DirectVolume::handlePartitionAdded, a memory corruption is triggered. Once triggered, it is possible to execute arbitrary code and gain root privilege. This was also referred to as *GingerBreak*.

3.3.1.6 Ashmem Access Issue

Ashmem is a mechanism by which memory can be shared between two processes. Due to an implementation issue, it is possible for local applications to cross the application sandbox boundaries and gain privileges [68]. Once higher privileges are gained, it is possible for the user to run arbitrary code on the device. This exploit works on Android versions prior to 2.3.

3.3.1.7 Solutions To Security Issues In The Kernel Layer

While it is trivial to roll-out fixes to the several exploits made public, it should be recollected that there are still 10% (as of Dec 2014) of Android devices that still run GingerBread (Android 2.X). This puts into perspective the gravity of these issues.

Another way to address such vulnerabilities is to implement something similar to MoCFI [69] in iOS application. Davi et al. [69] identify malicious behavior by identifying deviation in control flow of an iOS application. For each app, the authors statically build a control flow graph. This graph is stored in binary format and is validated at run time by a run time component. During run time, if any deviation is flagged to prevent the change of flow thus mitigating control flow attacks. The authors build a framework for iOS given its susceptibility to buffer overflow and return to libc attacks. A similar framework can be built for Android.

3.3.2 Libraries And Framework Layer - Security Issues Due To Implementation Flaws And Their Solutions

The libraries layer provides the basic C libraries and other third party libraries that were imported into Android OS. The framework layers provides the core of Android's functionality. These libraries provide the basic primitives that are needed to achieve SSL communication, 3D, 2D rendering capabilitis, storing data in databases etc... It is essential that these primitives do not have any vulnerabilities/implementation flaws. We present several implementation flaws that puts sensitive data to risk

3.3.2.1 Security Risks Due To Implementation Flaws In The Libraries Layer

1. Webkit

2. OpenSSL implementation

3.3.2.2 Security Risks Due To Implementation Flaws In The Framework Layer

1. Risks due to WebView's implementation flaws
   (a) Fracking
   (b) Code Injection
   (c) Improper implementation of trust base
2. Risks due to default export of Content Providers
3. Risks due to PackageManager's implementation flaws
   (a) Master Key Vulnerability
   (b) FakeID vulnerability
   (c) Privilege Escalation during upgrade

3.3.2.2.1 Webkit

Improper validation of floating point data in WebKit library used in Android prior to 2.2 allows remote users to run arbitrary code. It is also possible to cause denial of service by crashing the application via a specially crafted HTML document with a non-standard NotANumber (NaN) representation [70].

3.3.2.2.2 OpenSSL Implementation

Kim et al. [71] conducted a vulnerability analysis on the SSL implementation (OpenSSL library) in Android. If OpenSSL's pseudo random number generator (PRNG) can be predicted then malware authors can exploit this to leak sensitive data even if it is encrypted. Based on their analysis, they observed that it is possible to predict the PRNG due to boot time entropy hole of Linux PRNG. The boot time

entropy hole of Linux PRNG can be fixed by saving the PRNG before shutdown and loading it during boot time. This will thwart predicting the PRNG and keep all SSL sessions secure.

3.3.2.2.3 Risks Due To WebView Implementation Flaws - And Their Solutions

WebView [72] allows apps to show web content and behaves like a mini-browser. Yet, it does not provide the traditional security features as a browser does. Traditional browsers provide content isolation between applications where web-pages cannot access applications in the system. They also provide isolation based on origin. These security features ensure that untrusted web content is isolated from the general purpose applications. Since WebView does not offer these security features, it is vulnerable to attacks.

Luo et al. [73] created proof-of-concept attacks on WebView to demonstrate its vulnerability. They demonstrated two types of attacks - attacks on the app from the web-page and attacks on the webpage from malicious apps. These attacks were possible due to holes in sandboxing flaws in WebView.

In addition to the above attacks, Georgiev et al. [74] exposed vulnerabilities in third-party WebView frameworks such as PhoneGap. In addition to providing a WebView library, PhoneGap also allows access of local resources from a browser. This access is not protected by the same origin policy. This allowed attacks where a phone's local resources were accessed by a malicious webpage's code. They referred to these attacks as *Fracking*. To prevent such attacks, they proposed *NoFrak* which operates in a device-independent, platform-independent manner with no changes to existing PhoneGap based apps.

While traditional based apps are susceptible to attacks, Jin et. al conducted similar analysis on HTML5 based apps [75]. HTML5 based apps interact via several data

channels and hence should be susceptible to more attacks than traditional HTML web apps. They demonstrated how a person's location can be tracked and plotted on a map using a specially crafted 2D barcode image on a third party barcode scanning app. They also build a tool *NoInjection* [75] that provides countermeasures (by rewriting unsafe javascript API) to prevent such attacks.

These attacks clearly show how vulnerable WebView is against abuse of its features. To prevent abuse of benign app from a malicious webpage and vice-a-versa, it is essential that WebView provides browser-like security guarantees.

3.3.2.2.4 Risks Due To Default Export Of Content Providers - And Their Solutions

Until Android 4.2, the Content Provider component of apps was exported by default. If a component is exported, it is made available for other apps to use. This exposed all private data of the component unless the developer manually disabled it.

Zhou et al. [76] address this problem via the *ContentScope*. ContentScope identifies unintentional passive content leaks and content pollution. Content pollution occurs when an app's private data is polluted by a malicious app. ContentScope parses all the apps to identify the use of content providers. Once a set of candidate apps is selected, ContentScope identifies vulnerabilities by constructing a control flow graph. The authors check if an execution path exists to a publicly available content provider in the graph. If such a path exists, it can be accessed by malicious apps to leak/pollute the app's content.

3.3.2.2.5 Risks Due To PackageManager Implementation Flaws - And Their Solutions

PackageManager (PackageManagerService) is a framework module that is responsible for installing apps and updating the OS. Xing et. al. conduct a vulnerability analysis on PackageManagerService to identify possible vulnerabilities in the phone upgrade process [77]. It should be recollected that any vulnerability in the PackageManagerService is extremely critical as it is possible to gain control of critical system resources during the phone upgrade process. Based on their analysis, Xing et al demonstrated that it is possible to

1. Harvest permissions
2. Grab system resources by declaring system sharedUIDs
3. Contaminate data of old apps during upgrade process
4. Grabbing system resources and denying service to processes, users.

The authors build a tool *SecUP* to automatically detect such vulnerabilities.

BlueBox security [78] detected a vulnerability on Android OS starting 2.1. It was disclosed that Android does not verify the certificate chain of the app. This allows an arbitrary app to claim the identify of another arbitrary entity. Once a fake ID is achieved, the app is given the privileges of the victim entity.

Yet another critical vulnerability discovered in PackageManager is the Master Key Vulnerability [79]. Due to improper checks for digital signatures of applications, it is possible for attackers to execute arbitrary code. The vulnerability was due to the presence of multiple entries of AndroidManifest.xml file - where one entry is validated but the other entry is installed. This modification does not violate the cryptographic signature of the app but allows the attacker to execute arbitrary code.

3.3.3 Kernel, Libraries And Frameworks Layers - Security Issues Due To Design Flaws And Their Solutions

This section deals with security issues due to design flaws in the kernel, libraries and framework layers. A design flaw is a shortcoming that may have been overlooked during the design phase of a module/component/entity. Such a flaw could be exploited for malicious purposes and can be used to extract private sensitive information of the user. We present three major categories of design flaws

1. Design flaws that lead to abuse of kernel artifacts
2. Design flaws that lead to abuse of hardware
3. Design flaws related to permission model

3.3.3.1 Design Flaws That Lead To Abuse Of Kernel Artifacts

We present security issues due to design flaws in the kernel. These flaws lead to the abuse of kernel artifacts such as

1. Misusing telephony
2. Weakness in ASLR
3. Dynamic code loading
4. Abuse of /sysfs and /procfs

Poeplau et al. conduct a study of dynamic code loading techniques in Android [80]. Dynamic code loading is a common evasion technique for malware and is likely a source of vulnerabilities which can be exploited to load malware. Since Android provides several ways of loading code such as - DexClassLoaders, Runtime.exec, JNI

etc.. - the authors build a detection tool to detect dynamic code loading. To enforce integrity checks to prevent uncontrolled code loading, the authors suggest the use of application verifiers - which essentially "sign" the code and use whitelists to allow code loading. It is assumed that the application verifiers can be trusted.

A common attack vector in Android is the abuse of public resources within Android. Typical examples include /procfs, /sysfs, shared memory etc...

Chen et al demonstrated a mechanism to infer the content of the active application without peeking into the screen [25]. The authors track the usage of shared memory via tools available publicly in the /sysfs and /procfs partitions. In an offline training phase, training data is built with the activity transition graph and features (shared memory, CPU). In the online detection phase, these features are tracked by publicly available tools and are used to infer which activity is currently active. Once inferred, it is possible to hijack activity or to peek into the camera to gather sensitive data (check images)

Along the lines of [25], Lin et. al used the ADB proxy to "milk" Android screen for sensitive information [26]. To detect sensitive information, the authors built a database of CPU, memory usage, network usage, app workflow and app fingerprint information for each application. To reduce the size of the database, the authors built the database for only high value targets - such as banking apps. Once generated, the authors use the collected data to figure which app is currently running. Once a banking app is detected, ScreenMilker used ADB proxy to take screenshots of the app. Subsequently image analysis is applied to extract sensitive data.

Zhou et al. [27] try to glean a user's identity, location and personally identifiable information using a Zero Permission App (ZPA) - an app that requests no dangerous permissions at all! Instead they focus on Android's publicly available resources to reveal user's identity and location.

Firstly, the authors fingerprinted (and recorded) network usage of a set of apps to understand how each app works. Secondly, they use publicly available resources to observe network usage. Based on the network data fingerprint captured in Android's public locations, the authors infer which app was used. It is surprising that all information was inferred using only publicly available data. To mitigate these exposures, the authors suggest reviewing all publicly available data in Android OS. If any data has to be kept public, they suggest adding permissions to access such data.

Schlegel et al. [28] identified a new attack vector for stealing personal information. They demonstrated how credit card details can be extracted by using a sensory malware called *SoundComber*. The attack proceeds as listed below: The malware detects calls placed to banks and uses speech recognition signatures to identify the exact step in the Interactive Voice Recognition process where credit card information is input by the user. Once credit card information is extracted, it is communicated via covert communication channels such as changing vibration/sound settings, turning displays on/off etc to a colluding app. The colluding app uploads the details to a remote server thus completing the attack.

To address the threat of sensory malware, the authors implement a reference monitor which disables call recording when a predefined list of hot-lines are called. They also propose muting tone sounds while entering numbers on a keypad and isolating telephony via finer permissions.

3.3.3.2 Design Flaws That Lead To Abuse Of Hardware

We look at how USB channel can be used to cause irreversible damage to phones. USB channel attacks are commonly overlooked by researchers since an attacker has to gain physical proximity to the device. Once he gains physical proximity, there is

very little that can be done to protect the resource (asset).

Wang et al. [81] exposed vulnerability of smart-phones due to security weaknesses in USB connectivity. They demonstrated three classes of attacks: attacks from computer to phone, attacks from phone to computer and attacks from phone to phone. They demonstrated attacks that could reimage a phone or infect the phone with malware. Thus, it is clearly evident that leaving the USB channel unmonitored leads to irreversible loss of data. To mitigate these attacks, the authors propose that USB channel be monitored via firewall-like mechanism called USB firewall. Additionally, they also propose employing authentication mechanisms to establish a USB connection (similar to Bluetooth).

Aviv et al. [82] investigated gleaning identity, screen lock PIN and private data of the user using accelerometer data. The authors consider two attack scenarios - one where an attacker has a database of known usage patterns and the other where such info is not available. They use logistic regression algorithm to classify newly recorded pattern against patterns stored in the database. If a database of patterns does not exist, then the recorded pattern is split into smaller components and is run through hidden Markov models algorithms for classification. Using these algorithms, the authors were able to predict unlock patterns and PIN on repeated attempts with an accuracy of 73% and 43% respectively. They suggest restricting sampling rate of accelerometer and possibly disabling access to accelerometer during sensitive operations (such as key lock/unlock, entering PIN) to thwart these attacks.

3.3.3.3 Design Flaws Related To Permission Model

Security issues due to design flaws in the permission model arise due to

1. Permission Usage

2. Permission leakage

3.3.3.3.1 Permission Usage

In this section, we present common issues with permission usage.

1. Overprivileging apps

2. Native code privilege issues

3.3.3.3.2 Overprivileging Apps

Over-privileged apps pose a threat to security since malicious apps can exploit them to leask permissions. Peng et al. [83] recognized the need to rank risks/apps in an attempt to discourage over-privileging. This is similar to the User-Access Control prompt in Windows Vista/7/8+ and it's subsequent effect in making applications use less administrative features to prevent unnecessary UAC prompts (which affect user experience). The risk scoring function should be monotonic and should honor the principle of least privilege. It should give high scores to malicious apps and over-privileged apps. They use Naive Bayes machine learning method with information priors to build the scoring function.

During install-time, users are presented with a list of permissions that need to be granted without any contextual information as to how the permission is going to be used by the app. Pandita et al. [84] used natural language processing to read through the application's description to identify sentences which describe how the permission will be used. This will provide more context to the install permission screen. While more data about use of permissions is certainly welcome, chances are that this might confuse users more.

On the same lines of WhyPer, Qu et al proposed *Autocog* [85] which measured app description to permission fidelity. Autocog employed natural language processing to understand sentence structure and models the relation between descriptions and

permissions. The relation is built by analysis of app descriptions. While whyper compares an app's description with the developer documentation, Autocog compares one app's description with other apps' descriptions.

Gorla et al [86] proposed *CHABADA* a mechanism to check app behavior against app descriptions. They represented each app by a topic and cluster all apps (available in the marketplaces) into clusters based on the topic. Once the clusters are generated, they model permissions used in the cluster. They check app behaviors by recording deviation from the model to detect unadvertised behavior.

Au et al. [87] studied the usage of various documented and undocumented permissions (*PScout*) in Android framework. Using static analysis, they mapped API calls to permissions. They also constructed call graphs over the entire android framework and conducted a backward reachability search to identify entry points. Based on their analysis they found that there are very little redundancy in permissions. Additionally, permissions are not heavily connected. They also found that across successive versions, the number of permissions increases but is largely consistent with the increase in code size.

Complementary to other approaches, *VetDroid* checks and restricts the usage of the same once they are granted. Zhang et al. [88] propose *VetDroid* as a means to check and restrict usage of permissions. They identify points where apps exercise API guarded by permissions as Explicit Permission Use Points. They also identify places where resources held by permissions are used in the app. These points are referred to Implicit Permission Use Points. Any place where these two points intertwine is a good location for information to leak through. They model permission usage as a graph. Once the usage graphs are generated, a behavior profiler runs to search all the permission use graphs. The profiler discards graphs with no dangerous permissions and displays the rest of the graphs. VetDroid can be used by malware analysts as

it provides more contextual information about the use of permissions in an app. It can also help detect subtle vulnerabilities in apps.

3.3.3.4 Native Code Privilege Issues

Android allows applications to be developed using native code. Native code has the same privilege level as that of the apps.

Fedler et al [89] proposed ways of controlling native code execution in an attempt to prevent native code attacks on Android. They propose ways of monitoring the chmod syscall to prevent on the fly changes to executability of binaries, controlling executability of binaries at system level and other path based restriction mechanisms.

Ho et al [90] proposed a mechanism to contain malware infections on an infected device. To identify malware on the phone, *PREC* labeled all calls from third party native code. After labeling, a model is generated for benign apps and these models are stored in the cloud. On the phone, PREC dynamically identifies calls from native code and performs anomaly detection based on the models stored in the cloud. If a root privilege escalating malware is detected, then PREC can slow it down or possibly kill the malware thus containing the infection.

3.3.3.4.1 Permission Leakage

1. Content leaks

2. Capability leaks

3. Component hijacking

Lu et al. [91] identify component hijacking vulnerabilities via static analysis. The authors categorize permission leakage, intent spoofing and unauthorized data access as component hijacking vulnerabilities. The authors define six behaviors as

vulnerable behaviors and use them to identify vulnerable apps. Each app is analyzed to see if they exhibit any of the six behaviors. If they exhibit any one of them, then the app suffers from component hijacking vulnerability.

Grace et al. [92] addressed the problem of permission leakage. They define two types of leakage - explicit leak, implicit leak. An explicit leak occurs when an execution paths exists from the public interface to the desired capability (permission of interest) and if the public interface is not guarded by permissions and the execution path does not check permissions of unauthorized app. Similarly an implicit leak is said to have occurred when two apps sharing the same uid exist where one app's permissions are inherited by another app due to lack of permission checks for publicly available interfaces. The authors propose a static analysis method to identify such leaks. Analysis proceeded in two steps - possible path identification by constructing a control flow graph to identify all possible paths and impossible path pruning by eliminating paths that do not follow from previous system state. They conduct their experiments on stock apps - it may be recollected that stock apps generally have more categories of permissions at their disposal than third party apps. The leaks identified in stock apps can be exploited to reset the phone to its factory state. The authors discuss the placement of permission checks at the OS level or at the application level each with its benefits and drawbacks.

Bugiel et al. [93] addressed the problem of privilege escalation due to confused deputy attack and colluding apps. They propose a solution employing a system centric, policy driven run-time monitoring mechanism at the framework layer and a Mandatory Access Control mechanism at the kernel layer. These two layers interact with each other to enforce policies. The authors reused their previous work, XManDroid, as the framework component and added a new kernel level module. Other components include a reference monitor and decision engine. The authors tag all

intents using the uid and represent IPC as a graph. If a path traverses from one application with weak permissions to another application, it is tagged by the system as a privilege escalation attack and is thus thwarted by the reference monitor.

Felt et al. [94] address the problem of privilege escalation in an language independent, run-time independent and easy to use way. They introduce the idea of IPC inspection at OS level. For each IPC request, the upstream caller information is tagged along with the request. In the OS whenever an IPC request is made, the permissions of the requesting app is reduced to the intersection of the requesting app permissions and that of all of its callers. Thus confused deputy attacks and colluding app attacks can be prevented by this approach.

Marforio et al. [11] discuss the gravity of colluding applications. Android's security policy of granting permissions at install time applies only to the app being installed. It does not extend to two or more apps colluding with one another. In this work, the authors identify all the overt and covert channels that are available for malicious content authors to use. They also measure the bandwidth of each channel. The identified overt channels are SharedPref [95], Internal storage, external storage, logcat [96], socket communication and broadcasting intents. The covert channels include modifying phone's settings, thread enumerations, socket discovery, free disk space manipulations and processor frequency modification. The authors note that some of these channels are really hard to identify. For example, two colluding apps communicating via subtle changes in processor frequency are hard to identify and address. Some of the reported channels can be addressed by adding new permissions, limiting the availability of public data to apps. The authors suggest that these channels must be addressed at any one or combination of the following areas: during system design, at the application level, at the OS level and at the hardware level.

Nadkarni et al. [97] addressed the *Data Intermediary Problem*, an issue where two

colluding apps may accidentally leak/disclose information in their work-flow. Their work focuses only on accidental disclosure of data - not on intentional disclosure of data. The authors achieve this by allowing a developer to restrict the actions other apps can do in its work-flow. They provide two types of restrictions - export restrictions and required restrictions. Export restrictions dictate which other apps can send data off the host (phone). Required restrictions restrict the use of certain apps in the work-flow. The authors suggest that developers/users can decide which apps to restrict access to. Once a policy is built, the system makes sure that only those apps are allowed to interact with the developer's app. As the policy contains information on who can send data off the device, it addresses accidental disclosure of data. The only caveat being that developers/users should hash out all possible work-flows. Additionally, situations where a desired app is not available should be handled. Moreover, trusting an app once forever is not advisable since an app trusted now can become untrusted at a later point of time (due to a vulnerability discovered in the wild).

Zhang et. al. [98] proposed a mechanism to automatically patch the app's vulnerabilities without the intervention of the app's developer. They argue that security issues may not be a high concern for all developers alike. They may prioritize releasing new features in an attempt to stay/out-run their competition and may focus on security issues once they have established a size-able user base.

Patching an app should be done without adversely affecting performance and functionality of the app. There maybe minimal impact on performance but most importantly the vulnerabilities should be effectively disabled. The authors proposed *AppSealer*, a tool to automatically fix component hijacking vulnerabilities in apps. To patch a vulnerable app, AppSealer requires as input an identified vulnerability (identified via CHEX [91]). AppSealer uses tainting to identify data leaks and program

slicing to identify program slices where the vulnerability exists. Once a vulnerable program slice is identified, propagation of taint is tracked and patch statements are added to prevent access to private data. In order to reduce the overhead due to patch statements, the authors propose a series of optimization steps. Once the patch statements are optimized, the vulnerability is patched and the app is repackaged. Repackaging the app essentially removes the original digital signature of the app. This may be overcome by caching a copy of the signing key in the app marketplaces and/or returning the app to the developer for digitally signing it.

Yang et al. [99] take a different approach to the same problem. They note that not all data transmission leaks are malicious. They define a data transmission leak to be malicious only if the user did not intend the data to be transmitted. In their work, *AppIntent*, they propose mechanisms as to how to capture user intentions and how to identify data transmission leaks. By using static analysis, the authors identify all places where transmission of data happens. By leveraging a event space constrained symbolic execution, they extract all critical events and essential events. During this dynamic analysis, they automatically trigger event input, data output, highlight activated views of the user interface and thus identify the sensitive data being read and transmitted. These leaks can help an malware analyst to understand more about the offending app.

3.3.4 DAC Design Limitation - New Security Architectures

Complementary to these solutions, several researchers proposed new security architectures in these critical layers. Some of them are

1. SEAndroid

2. FlaskDroid

3.3.4.1 SEAndroid

Android kernel currently employs Discretionary Access Control model for ensuring security. Smalley et al. [100] suggested the use of Mandatory Access Control (MAC) in Android OS. MAC provides system wide guarantees for security policies which cannot be overridden by users intentionally or accidentally. It provides stronger isolation and sand-boxing of apps, confines damage by background daemons, flawed and malicious apps (even with root permissions) and supports centralized policy configuration. Configuring policies from a centralized location/entity, brings flexibility to the policies being enforced. In their prototype *SEAndroid*, the authors implemented MAC protocol in Android. At the kernel level, ashmem and binder were modified. Additionally, the file system had to be enhanced for supporting extended file attributes. At the framework level, Zygote was modified to support MAC. Some of the patches developed by the authors are currently being integrated into the Android kernel and user-space levels. Bugiel et al. [101] extended SEAndroid to support access control on multiple layers, enhanced context awareness and multiple stakeholder policies. They used SELinux's type enforcement to support apps, intents, content providers and multiple stakeholders. They use SEAndroid for kernel level MAC support. They implement framework level modules to enforce all policies. Policies allow definition actions on individual actions inside a coarse permission. Thus, fine-grained control in addition to the flexible MAC (from SEAndroid) is achieved.

3.3.4.2 FlaskDroid

Bugiel et al proposed a mandatory access model based on the Flask framework [102]. The Flask framework allows decoupling policy from policy enforcement architecture. FlaskDroid is built on top of SEAndroid using a custom policy language

and userspace security servers. Using learning/semi-automatic mechanisms, policies are generated. In addition to these policies, FlaskDroid also allows application developers to define their own policies. Policies are enforced by the userspace security server.

3.3.4.3 Android Security Modules

Heuser et al proposed *Android Security Modules* [103] akin to Linux Security Modules of Linux to promote OS security extensibility. Their mechanism allowed reference monitors in the app layer. Reference monitors register for sensitive operations. When a sensitive operation is executed, the registered callback is invoked and the reference monitor ensures that the sensitive operation is allowed to go through. These reference monitors allow for complete mediation, verifiability and tamper-proofness.

Fig. 3.5 presents an overview of available solutions to security issues in these layers

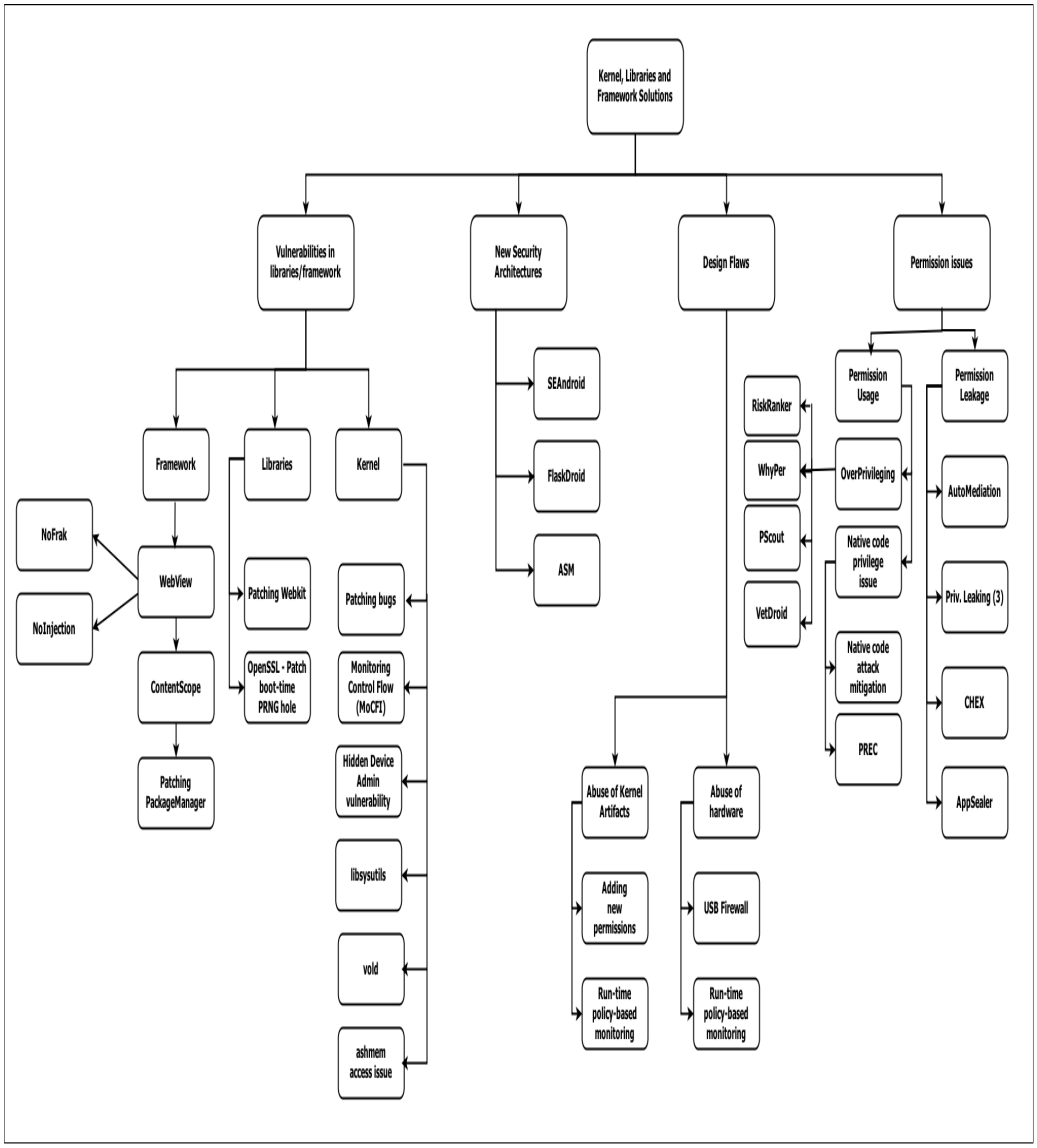

Figure 3.5: Solutions to Security challenges in the Kernel layer

4. SYSTEM DESIGN

4.1 Threat Model And Trust Model

4.1.1 Threat Model

In this section we discuss the threat model and the trust model for USB-Droid. Our threat model consists of two applications/components of applications - referred to as entity 'A' and entity 'B'. Entity 'A' is a malicious entity which seeks to extract personal information of the user. Entity 'B' is a benign/malicious system/user entity that has the capability of transmitting data to remote servers. For instance, 'B' could be an app with INTERNET permission or could be the system browser provided by Android. The phone stores personal data 'D' of the user. We assume that a skilled adversary has deployed both 'A' and 'B' (in case it is a user entity) onto the phone. We also assume that 'A' has successfully managed to extract the sensitive data 'D' from the user. Now, 'A' wants to send this data out of the phone via 'B' and can transmit this data to 'B' via any of the covert channels of communication (Figure 2.1). We intend to block the communication via the channel and thus prevent the data from leaving the phone.

4.1.2 Trust Model

For the purposes of our system design, we assume that the frameworks, libraries and the kernel layer of the Android device are not compromised. We also assume that the device does not have root privileges. This assumption is fully valid given that all hardware vendors disable root privileges on devices before shipping them out. Root privilege can only be enabled after installing the 'su' binary on to the phone. We also assume that the phone has several other apps (real-world scenario)

that the user has installed or has several stock apps that come pre-installed with every phone.

4.2 Motivation

We examined covert channels in the Android framework [12], [11] and noticed that most of the channels fall under the Binder's infrastructure. We define the Binder infrastructure comprises the Binder kernel module (/dev/binder) along with the ServiceManager component. It also comprises the tokens and the protocol that Binder uses to communicate with processes. We also observed that the covert channels are (mostly) synchronous in nature. A channel is synchronous if both the sender and receiver need to be running simultaneously to transfer data. Only a few channels do not use Binder for communication - however they can also be brought under Binder's infrastructure for monitoring purposes. Thus, the Binder layer is an appropriate place for solving the issue of malicious communication via covert channels.

If USB-Droid detects that a particular pair of apps use a covert channel and the channel bandwidth usage is greater than 100 bps [29], the apps are suspicious of colluding with each other. The value of the channel bandwidth depends on several factors such as Noise and Delay, Coding and Symbol Distribution, Measurements and Scenarios of Use and system configuration/initialization dependencies. Once we identify suspicious apps, we can check the transaction against security policies to confirm our suspicion. Once a suspicious transaction is identified, then we can notify the user of the ongoing transaction. The user may allow/deny the transaction based on the context where the app is running.

4.3 Overview

4.3.1 Tracking Use Of Covert Channels

As was noted in Section 2.3, a covert channel has limited expression - mostly one bit at a time. Hence, colluding apps repeatedly use the channel for transmitting data from sender to receiver. This leads us to note that our mechanism must handle the repeated use of channel on a per-app basis.

Also, we noted that the minimum bandwidth for a covert channel to be effective is 100 bps [29]. In case the observed bandwidth is less than 100 bps, the channel is no longer effective. If the bandwidth is larger than 100 bps, then the channel becomes an effective channel for communication. Hence, we can weave a threshold based mechanism to reduce the False-Positives due to our system.

In Section 2.3, we also noted that these channels only support one value at a time. Writing a new value overrides the old value. In order to properly identify the pair of colluding apps that use the channel, it is necessary that the receiver reads the exact same data as the sender. Thus, for each app using the channel, we need to record the last value that it wrote and also track the current value in the channel (See Figures 4.1, 4.2).

Table 4.1: Table template to track app-channel usage

AppName	Last Value	Last Operation	Current Value	No. of times channel was used	Latest use timestamp
....	....	....	.....	....	...
com.example.app1	Value_1	Write	Value_1	Count_1	Timestamp_1
com.malware.app1	Value_2	Write	Value_2	Count_2	Timestamp_2
com.example.app2	Value_2	Read	Value_2	Count_3	Timestamp_2
....	....	....	.....	....	...

Table 4.2: Tracking app-channel usage - A concrete example

AppName	Last Value	Last Operation	Current Value	No. of times channel was used	Latest use timestamp
com.example.app1	1	Write	1	44	00:00:01
com.malware.app1	0	Write	0	56	00:10:11
com.example.app2	0	Read	0	55	00:10:11
	app2 read the value written by malware app1				timestamps are also similar

Table 4.2 shows an example where a covert channel is used for malicious communication. The covert channel being used is the enabling and disabling of vibration settings. This is a common method where a colluding app enables the vibration settings to indicate a '1' (bit 1) and disables it to indicate a '0' (bit 0). At 00:00:01, com.example.app1 enables vibration settings. We create a new entry if one does not exist - otherwise, we update the entry with the latest value written. We note if the operation is a Read/Write and also note the current timestamp.

Later on, app com.malware.app1 writes a zero to the vibration settings. Once that is complete, a possibly benign app com.example.app2 reads the zero from the vibraton settings. However since the number of times this channel was used has not crossed the threshold (55/56), we do not match policies. When the usage crosses 100 bps, then we start to match transactions with the policies in the system.

4.3.2 USB-Droid Modules

A high level working of the USB-Droid is shown in Figure 4.1.

We introduce several new modules and modify existing ones to achieve our pro-

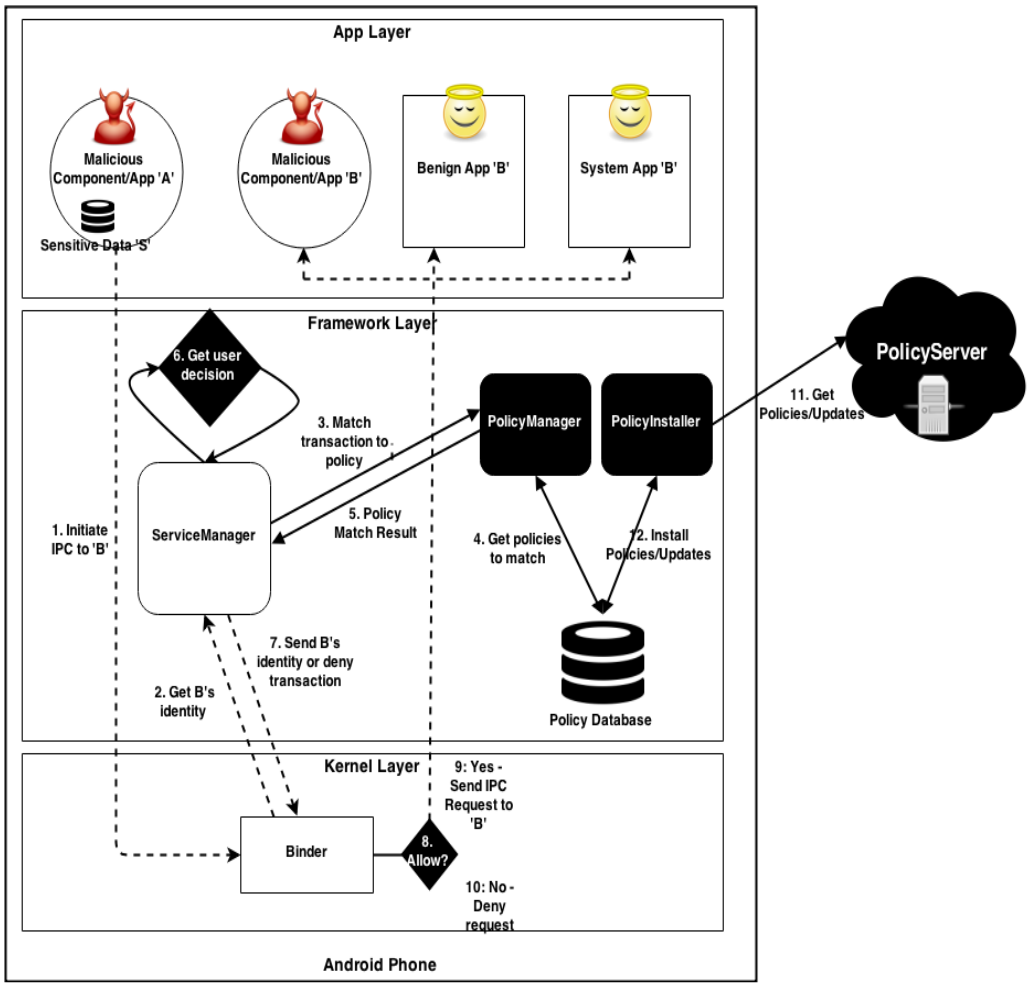

Figure 4.1: Proposed Universal Skeptic Binder-Droid

posed model. A summary of these modifications are provided below.

4.3.2.1 Newly Added Components

1. PolicyManager: Interface to the PolicyDB and also matches transactions to policies

2. PolicyInstaller: Installs new/Updates existing policies into the PolicyDB

3. PolicyDB: the SQLite DB storing all the policies

4. PolicyServer: Serves policies to all clients

4.3.2.2 Modified Components

1. ServiceManager: Modified to check for policies matches and consult the user for action to be performed

A typical end-to-end IPC work-flow in USB-Droid is now as follows: Malicious component A attempts communication to B to transfer D. The request goes through the Binder module. Binder sends the request to ServiceManager to retrieve Bs identity. ServiceManager sends the request to PolicyManager to check for matches. PolicyManager has access to the PolicyDB and checks if the transaction matches any policies in the DB. The result from the matching process is sent to the ServiceManager. If a policy matches, we show a pop up to the user asking him to decide what needs to be done. The user can choose to allow/deny the transaction. Depending on user input, ServiceManager sends Bs identity or denies the transaction. Depending on ServiceManagers output, Binder may allow the transaction (Step 9) to complete or deny it (Step 10). Periodically, the PolicyInstaller can get updates from the server and install them into the database (Step 12).

4.4 USB-Droid Components - Detailed Description

In this section, we detail the design for each component in our system.

4.4.1 PolicyManager

The PolicyManager module is responsible for managing the policies for USB-Droidand matching policies to incoming transactions. Internally, the PolicyManager also stores the bit vector representation for each app installed on the phone. Additionally, this module serves as an interface to the database. When a transaction using the covert channel is initiated, the PolicyManager matches the transaction to

any of the policies in the database. If a match is found, it informs the ServiceManager about the possible match. Once the ServiceManager receives the verdict from PolicyManager, it will proceed to request the user for a response.

4.4.2 PolicyInstaller

The PolicyInstaller module is responsible for installing policies onto the PolicyDB. The policies may be new policies sent to the phone during an app's installation or may be updates to existing policies. When the PolicyServer sends policy updates to the client, this module receives the updates and installs them on to the database. The incoming policies are in XML format and need to be converted to USB-Droid's internal representation. This details of the conversion process is detailed in 4.5.1.

4.4.3 PolicyDB

The PolicyDB is a traditional relational database powered by SQLite framework. The database stores USB-Droid policies in their internal representation. The database is also indexed to provide faster results to queries. The index chosen is the tuple of sender details and the channel of covert communication.

4.4.4 PolicyServer

The PolicyServer module is a traditional server that serves policies over HTTPS connection. We believe that with the adoption of this proposal in marketplaces, the market places will provide such servers. Policies are fed into the server by means of manual/automated static/dynamic analysis of apps.

4.4.5 Modifications To Service Manager

When an app launches a Binder request to communicate with another app, Binder sends a request to Service Manager to resolve the identity of the recipient app. The Service Manager stores a registry of IPC entities and their identities (like a key-value

map). This registry is built at boot time with every service notifying the Service Manager of their presence and identity. When it receives a request from Binder, it retrieves the recipient's identity and returns it to the Binder module.

However, we hook this process to support policy enforcement before returning the recipient's identity to Binder. We extended Service Manager to contact the PolicyManager with the sender, receiver and the channel details. The PolicyManager will match the incoming transaction based on the channel being used and inform the ServiceManager about a possible match. The ServiceManager will then launch a user-interface to alert the user about the suspicious transaction. If the user confirms the transaction, then ServiceManager will let it through; else it will be blocked.

4.5 Challenges To Address

Two major challenges that arise with our design are the following:

- Policy specification and method of enforcement
- Overhead due to policy enforcement

In this section, we look at how we modeled our solution to address these challenges.

4.5.1 USB-Droid Policy Syntax And Format

4.5.1.1 USB-Droid Policy Syntax

We represent our policy as a 3-tuple consisting of sender information, receiver information and the channel ID. The senders and receivers in the policies are identified by the intents and the permissions they use. The channel ID is a unique ID that has been created statically for each of the supported communication channels.

This method of representation is not common in Android. Android uniquely identifies entities by their package names. However, in USB-Droid, we represent

them by their permissions and intents that they use. Choosing to represent the sender and receiver by their package names can lead to flaw where the system can be bypassed by renaming the package name. Our scheme of representation prevent this way of bypassing USB-Droid. Also, similar software use permissions of similar kind. For example, social networking apps use the permission to access location, user details commonly. By representing these apps by the permissions that they use, we can handle new similar software that are published to the market place with little or no modification to the policies. This makes specifying policies extremely easy without the need to have highly specific policies. However care should be taken to not specify highly general policies as they might lead to False Positives.

4.5.1.2 USB-Droid Policy Format

To ease the adoption of our system, we decided to format our policies using the XML language. XML is extremely easy to understand and is highly flexible for security administrators to adapt to. A generic model of the policy is shown in listing 4.1.

```
<?xml version="1.0" encoding="utf-8"?>
<policies count="1">
<policy_name name="friendlyName">
<entity_A name="entityHumanFriendlyName-1">
<invokesIntent>
<intent_name> intent_name_1 </intent_name>
</invokesIntent>
<hasPermission>
<perm_name> permission_name_1 </perm_name>
</hasPermission>
</entity_A>
<entity_B name="entityHumanFriendlyName-2">
```

```
<invokesIntent>
<intent_name> intent_name_2 </intent_name>
</invokesIntent>
<hasPermission>
<perm_name> permission_name_2 </perm_name>
</hasPermission>
</entity_B>
<channel id="PREDEFINED_CHANNEL_ID" />
</policy_name>
</policies>
```

Listing 4.1: XML representation of a policy

It can be immediately observed that the policy can be defined with extreme ease. Also, it should be noted that multiple policies can be defined in the same XML document as long as each policy is unique.

4.5.2 USB-Droid Policy Internal Representation

While specifying policies in XML format eases the specification part it makes enforcing them harder. It is necessary to make policy matching as fast and efficient as possible. However, it is also important to make policy specification as easy and as flexible as possible. To address these seemingly conflicting goals, we decided to decouple the representation formats for policy specification and policy enforcement.

To make policy enforcement efficient we represent sender and receiver using a bit vector. We compile a list of standard Android intents and permissions and create an ordering of these into a bit vector. For each entity represented in the XML policy, we can now generate a bit vector. While installing the policy to the database, the PolicyInstaller will generate this bit vector using the following approach. A bit in the bit vector is set if the corresponding permission/entity is set in the XML policy.

The bit is set to zero otherwise. Thus the entities are represented internally as bit vectors and the channel of communication is represented as an integer data type.

This representation gives us a very fast matching method as we will detail in Chapter 5.

5. SYSTEM IMPLEMENTATION

In this section, we describe the implementation details of USB-Droid.

5.1 Android AOSP Branch

We implemented our system in Android 4.4.4 release 2, codenamed Android KitKat. This system was the latest at the time we were implementing our system. The latest version, Android Lollipop (Android 5.0) was released later on and we have plans to port our code to Lollipop as well.

5.2 USB-Droid Components' Implementation

5.2.1 PolicyManager

The policy manager interfaces with the database to match policies to incoming transactions. The policies are stored in a database in the kernel layer. The policy manager has been implemented in two layers - one in the kernel layer and the other in the libraries layer. This was necessary since amongst the covert and overt channels that have been identified, not all pass through the binder layer. To accommodate all the channels, we implemented a policy manager in the libraries layer where a few covert communication channels exist.

It may be recollected that Binder transactions are routed to Service Manager (service_manager.c) for resolving the recipient identity. For channels that use the Binder protocol, we hook the appropriate functions in service_manager.c to invoke the policy manager. The hook will provide information regarding the sender, receiver and the channel being used. The policy manager will receive these information and will query the database for matching policies. The database is indexed with the channel information, therefore we will only retrieve the appropriate subset of policies

to match with the current transactions. For each policy returned from the database, we compare the sender info and the receiver info to identify matches. The sender and receiver information are stored in a bit vector and the comparison is effected by means of bitwise operations - bitwise AND operation. Thus policy matching can be done in linear time of the number of policies.

For the channels that do not use the Binder module - viz File operations, Network operations, operations on the /proc/ and /sys/ channels - we adopted a different approach. For such channels, we implemented an interface to the policy manager in the libraries layer. When any of these operations reaches the library layer, they are redirected to the policy manager. The policy manager is fed with the sender, receiver and the channel information. It is to be noted that, we bypassed the actual binder driver and reached the policy manager. This is so as to prevent an unnecessary binder transaction and a redirection to the service manager. Since we directly reach the policy manager, the manager can fetch the policies from the database and check for matches. Thus we reduce the load on the Binder module and increase efficiency by avoiding a costly Binder transaction.

5.2.2 PolicyInstaller

The policy installer is the module directly responsible for installing the policies into the database and also communicating with the policy server via a secure channel of communication. The policy installer in our system communicates with the policy server via the SSL channel (port 443). The policy served by the PolicyServer is in XML format and needs to be converted to USB-Droid's internal representation for efficient operations. Firstly, the policy is validated for proper XML form and then processed. Each policy has a list of intents and permissions. Using these lists, we convert the policy to a bit-vector based representation where the sender and receiver

entity is represented by a bit vector. The channel is represented by an integer. To aid in the conversion process, we compiled a list of intents and permissions that are official supported by the Android device and statically assigned it to a position in the bit vector. While processing the policy, we set the bit corresponding to the permission being used by the app to 1 and 0 otherwise. This leads to the limitation that we cannot support custom permissions. Given that majority of the apps use default permissions, this is not such a big limitation. Support for custom permissions can be extended trivially. We implemented a helper module to help with the SSL communication with the server. The bit vector representation of entities is to make policy matching really simple as has been explained in previous subsection.

5.2.3 PolicyDB

The PolicyDB is a SQLite database provisioned inside the kernel layer. The SQLite amalgamation library is cross-compiled with the libc library of Android (also existing in the Android Open Source Project sources). To ease database operations, we wrote a wrapper to interact with the SQLite library. A private instance of the wrapper is instantiated inside the policy manager/installers to ensure that only those modules have access to the database. Thus, by virtue of object boundaries, this database is protected from modification. Also, as we assumed that the device does not have root privileges (real-world scenario), the database cannot be modified by any malicious/benign process accidentally/intentionally. Thus the system's integrity is maintained.

5.2.4 PolicyServer

The PolicyServer is a central off-line module that serves policies to all USB-Droid clients. In the Android ecosystem, these PolicyServer modules may exist in each of the market places or there could be a centralized policy server for several

of existing market places. Policies defined by security administrators are served to the clients via SSL channel (port 443). Ideally, to reduce the overhead of polling, the server will support PUSH notification system so that clients can receive updates directly rather than frequently poll for updates.

5.3 USB-Droid Use-Cases

5.3.1 Policy Installation

Policy installation is a process where the security administrator defined XML policies are translated into USB-Droid's internal representation. The installation process is trigged by the installation of an app. We modified the PackageManager module - the module that is responsible for installing apps/system updates to the device - to trigger the download of policies. During installation of an app, we request the server to see if there are any updates. This request is done in an app-agnostic manner. If there are any updates on the server, they are downloaded by the PolicyInstaller module and converted to USB-Droid's internal representation. By using pre-compiled lists of standard Android intents and permissions, the policies are mapped to bit vectors. Thus the entities are represented internally as bit vectors and the channel of communication is represented as an integer data type.

5.3.2 Policy Update

In addition to the above installation process, the PolicyInstaller is capable of receiving updates via a PUSH mechanism and can also poll for updates. When a new policy has been defined on the server, the PolicyServer will notify the PolicyInstaller about the availability of a new policy. Alternatively, the PolicyInstaller is also capable of frequently downloading policies from the server and installing them on the phone. Once the policies are downloaded, they are converted to the internal representation and stored in the database.

5.3.3 Handling Transactions

Malicious app communication can be effected by overt or covert channels. Also, we have previously pointed out, some of these channels do not fall under the Binder infrastructure. For those channels, that go through the Binder infrastructure, we handle them by modifying the service_manager.c to check for matches and inform the Binder driver of the match verdict. Service Manager will track the usage of apps and channels as detailed in section 4.3.1. The service_manager.c module communicates with the framework level module to show an UI to the user. The user must respond to this by choosing to allow/deny the transaction. If the transaction does not go through the Binder, it will be picked up by the libraries layer USB-Droid module. That module will contact the policy manager and check for policies and the rest of the steps mimic the kernel level check. The user is notified for confirmation and then the transaction is allowed to go through or stop.

Due to the above explained mechanism and design, it is for the first time that we are able to handle both native and Java based apps using the same security solution. This is non-trivial due to differences in the work-flow of Java based and native apps. Since we operate at the lowest layer - the Binder layer - we can effect policies on apps that use of the mechanisms. Thus, native code based malicious software can be handled with the same ease as that of Java based malicious software. This is the biggest advantage of our solution - and thus the unique proposition to USB-Droid.

6. SYSTEM EVALUATION

For our system to be effective at preventing malicious app collusion, the overhead due to our system should be low and the policies must be effectively flagging malware that collude over covert channels. We've implemented the core modules of USB-Droidon Android KitKat 4.4.4 revision 2. The PolicyManager and PolicyInstaller modules have been implemented in C/Java and are integrated into the ServiceManager modules. The PolicyDB is realized by means of SQLite database built alongside with the USB-Droid modules. The PolicyInstaller can contact a remote server and request for policy updates to the system. Once new policies are downloaded, they're stored in the PolicyDB. For each Binder transaction, the ServiceManager will perform policy matching by retrieving policies form the DB. It is utmost critical that the overhead due to USB-Droid remains as low as possible. We ran our tests on a Nexus 5 phone and a test pool of emulators. We measured the overhead incurred by USB-Droid and also to measure the effectiveness of enforcing policies at the Binder layer.

6.1 Overhead Due To Policy Enforcement

Since USB-Droid modifies core component of Android's IPC mechanism we evaluated the overhead added due to our system. We evaluated our system against several Binder transaction sets. Each set consists of a varying number of Binder transactions chosen randomly from a pool of all possible Binder transactions. The number of transactions varied from 100 to 1000000. We ran these sets multiple times to get an average value of overhead incurred on a per transaction basis. Based on our evaluation, we observed *an overhead of 0.01145 milliseconds per transaction.* It can be observed that our system has very little overhead and hence provided empirical

basis for our claim that it is a light-weight solution.

One interesting observation we made during the evaluation was that until 10000 iterations, the Dalvik Virtual Machine does not optimize the code. However, beyond that DVM optimizes the code and converts it to native code. It is common knowledge that native code runs significantly faster than Java code - thus our system gets faster as the number of transactions increases.

6.2 Boot Time Impact

During our evaluation, we noticed that the system boot time is a Binder intensive operations. During boot time, several hundred Binder transactions are initiated by the system and it is important that USB-Droid does not delay the boot process significantly.

Since the system has not completely booted, it would not be possible for a UI to be displayed asking for user confirmation. Thus, during the boot process, we do not interfere with the Binder process. Our system will still perform policy matching and allow the transaction to proceed irrespective of the outcome of policy. This would not allow any malware to creep in because of two reasons. One, no user entity is allowed to run before the system is completely booted up. When the system has completed the boot, it sends a broadcast message to all apps that they can schedule their operations. To run malicious code before this intent is sent, the kernel has to be modified to run malicious code or the phone has to be rooted to hook in malicious code. Since we assumed that the phone is not rooted and that the kernel is trusted, overriding our mechanism does not lead to malware bypassing our system.

We evaluated the system boot times of the Nexus 5 phone with and without USB-Droid. We obtained the default Nexus 5 KitKat factory images from Google's website and compiled a list of apps that we pre-installed on it. Since this is the factory

image, there were the stock apps (Gallery, Email etc...) and Google's app package (comprising of GMail, YouTube, Google Play Components etc...). We repeatedly powered down the device to a complete halt and booted the phone from this cold start state. This was repeated for ten times. After this process, we installed our custom ROM onto the device and installed all the Google apps that were in the factory image. This was done so as to create a uniform install base to reliably measure boot times. After this process, we followed the same procedure of boot time measurement mentioned previously.

The Nexus 5 phone with Android 4.4.4-r2 OS boots in an average time of 17.6 seconds whereas the same phone with USB-Droid boots in an average time of 18.9 leading to an average delay of 1.317 seconds per boot. This translates to a 7.49% delay in boot times.

6.3 Benchmarks

Using the Antutu performance evaluation tool, we established benchmarks for the phone with and without USB-Droid. Antutu is a commonly used benchmarking tool which conducts a wide range of operations that measure the phone's performance. Common operations include integer operations, floating point operations on CPU, RAM speed, IO and GPU performance in rendering 2D and 3D graphics. We ran Antutu on the default factory image for ten times and observed the average values. After that, we installed the custom ROM with USB-Droid and observed the average values across ten trials. Based on our evaluation, we calculated the average values and present them in table 6.1.

Based on our evaluation, we observe that the benchmarks with our system has decreased by about 6.6%. The benchmarks also show that our system produces a fully usable system that has a slight performance overhead of less than 7%.

Category	Operation	Without USB-Droid	With USB-Droid
UX	Multitask	4575.4	4426.4
UX	RunTime	2209.6	1864.4
CPU	CPU integer operations	3233.2	3096.6
CPU	CPU floating point operations	2927	2593.6
CPU	Single Thread integer	2564.8	2391.8
CPU	Single Thread floating point	2203	2137.2
RAM	RAM operations	1267.4	888.6
RAM	RAM speed	1588.2	1284.4
GPU	2d Graphs	1631.4	1630.8
GPU	3d Graphs	11623.2	11237.2
IO	Storage IO	1699.2	1583.8
IO	Database IO	645	636
	Grand Total	36167.4	33770.8

Table 6.1: Antutu Benchmarks with and without USB-Droid

6.4 Evaluation With Real World Apps And Malware

We evaluated USB-Droid and it's effectiveness via two types of testing - negative testing and positive testing. In negative testing, we ran our system on popular and trusted apps. This test was aimed at identifying how our system would behave with general day applications. This would also help us understand the False Positives picked up by the system. In positive testing, we implemented colluding malware from [25], [26], [27] and [28]. In addition we also implemented colluding malware using the ideas from the above papers on other channels.

6.4.1 Negative Testing

We evaluated USB-Droid using 100 different apps. Around 70 apps came preinstalled in the factory image of Nexus 5 device and the rest of them (30) were installed from Google Play Store. These apps were chosen from the top 3 in ten different cat-

egories. For our evaluation, we built a test bed of system apps, third party apps and malware. All apps were sanity tested in the course of the normal day operations. During our testing, we observed that there were no false positives on the system. This is because benign apps generally use overt channels for communication. Covert channels are generally used by stealthy malware. Hence, not having false positives is expected in this scenario.

6.4.2 Positive Testing

We evaluted USB-Droid against colluding malware built based on reports from [25], [26], [27] and [28]. We built colluding malware based on their report of how they executed their attacks. However, not all colluding channels were covered by the afore-referrenced work. Hence, we implemented similar malware logic on the uncovered channels so as to ensure a complete evaluation of our system across all covert channels. The entire list of malware that we developed are as below.

1. An application to steal user contacts by writing them into the system logs
2. An application to steal user identity and location by using the /proc, /sys channel
3. An application to steal user location via the single/multiple changes to system settings
4. An application to send custom sensitive information via changes to vibration settings. Custom information was stored in an app's private storage. Since the malware was packaged with the app, we could access that information.
5. An application to send multiple bytes of custom sensitive information via enumerating threads. This is an entire malware application.

6. Two malware applications that communicate via discovery of UNIX sockets.

7. Two malware applications that communicate via changes in disk free space.

8. Two malware applications that communicate via periodic broadcast intents for gradual discharge of battery.

9. Two malware applications that communicate via the last placed call.

We now discuss a few policies that we developed during our evaluation in tables 6.2, 6.3 to flag these malware.

6.4.2.1 Policy To Detect Leakage Of User Identity And Contacts Info

In this example, an app which has access to the user's contact information colludes with a weather app which has permission to access the Internet. The two apps collude via the logs channel. The sender app writes sensitive contact information to the logs and the weather app can read that info from the logs. To prevent this communication, we defined the following policy.

Entity	Permissions
	android.permission.READ_CONTACTS
	android.permission.GET_ACCOUNTS
Receiver	android.permission.INTERNET
	android.permission.ACCESS_NETWORK_STATE
	android.permission.ACCESS_WIFI_STATE
	android.permission.READ_LOGS
Channel	LOGS

Table 6.2: Policy to identify communication via LOGS channel - Contact Stealer malware communicates with Weather application to steal user contacts [11]. Similar policies exist for other covert channels

6.4.2.2 Policy To Detect Leakage Of User location, Health Related Info

In this section, we discuss how we detect the theft of user location via an app that does not use any permissions. The attack vector is as follows: The attacker app 'A' uses the /proc and /sys channels to gather information about app's operations. An example would be accessing the Internet usage history. This information is world readable on Android phones. Once the history is accessed, the app uses predefined fingerprints to identify which app is being used and what functionality of the app is being exercised. Once the apps are identified, the authors use other external tools (such as Twitter APIs) to ascertain the user identity.

We defined the following policy to detect this threat. It can be seen that the sender does not have any permissions but we're still able to detect the colluding transactions. This is because we monitor the covert channels and track their usage. When an app colludes maliciously, our system will identify the misuse and flag it. In this case, the covert channels are used extensively and our system correctly identifies them. In addition, we see that the receiver app has sensitive permissions that allow it to send data to an offline server. Since our policy captures that, we're able to identify such misuse with ease.

Sender	No permissions
Receiver	android.permission.INTERNET
	android.permission.ACCESS_NETWORK_STATE
	android.permission.ACCESS_WIFI_STATE
Channel	PROC_SYS

Table 6.3: Policy to identify zero permission app stealing user location, identity and personal medical info

In our evaluation, we conducted two separate trials that show the effectiveness of our system. In the first trial, we override our system to allow all Binder transactions irrespective of the policy matching results. We used the phone under this setting for about a day - sending messages via the default app, sending texts via messaging apps, watching videos on YouTube, sending emails via the GMail, Yahoo Mail apps, checking on weather via Yahoo Weather app and listening to music via music streaming apps. Due to our light-weight solution, the evaluator would not notice any difference with the addition of USB-Droid. In another trial, we override our system to deny all transactions irrespective of the policy matching results. This rendered the phone useless as the phone would get stuck during the boot process. Since there are a lot of Binder transactions during the boot time, any failure to complete them raises an exception and thereby hangs the boot process. Thus, it can be seen that operating on the Binder layer is indeed very effective for providing security mechanisms.

6.5 Benefits Observed

Based on our evaluation, we observe that USB-Droid is a very light-weight yet effective solution. Since we're using XML language to define our policies, our policy language is really easy-to-learn. The overhead due to policy matching is reduced because of our internal bit vector based representation and linear time scan for policy matching.

7. DISCUSSION AND FUTURE WORK

The following chapter discusses the limitations of our work and also suggests future work that can be done to improve the system.

7.1 Discussion

In Chapter 4, we detailed the mechanism of creating a bit vector out of the incoming XML policy. In that we mentioned that we compile a list of standard permissions and intents supported by the system. As of the prototype, we do not support custom permissions. It is possible for Hardware Vendors and App developers to add custom permissions. When a new permission is added by a hardware vendor then it will still be present in the list we compile. However if a custom permission is added by an app, then we at the present do not different the individual permissions. We only record the presence of custom permission in a bit. However this can be easily modified to differentiate between custom permissions by assigning each one a separate bit and regulate the allocation on the server side. However, the prototype does not have this functionality. It should be noted that this is not a limitation of the design but of the implementation.

Secondly, it is important to define policies appropriately. Defining a very coarse policy will lead to benign transaction being flagged as malicious and thus increase the number of False Positives. On the other hand, defining a very fine policy may lead to malicious transaction being neglected as benign transactions and thus may increase the number of false negatives. Since the policies are defined by the security administrator, it is assumed that the administrator understands the consequences of loosely defined policies.

7.2 Future Work

As detailed in the previous section, policies are manually defined. This gives the security administrator sufficient flexibility in defining what policy needs to be enforced. However as the number of apps keep increasing, manual efforts may not scale well. We have plans to implement a machine learning algorithm to define policies automatically. The system will record all transactions based on static or dynamic analysis of app code. Once the transactions are recorded, a supervised/unsupervised algorithm can classify transactions as malicious/benign. Based on the classification, policies can be extracted by the use of association rules. The extracted policies can be installed on the phone (after manual verification/testing if need be). We are presently conducting a feasibility study to achieve this.

8. CONCLUSION

Ever since the first release of Android, it has grown its user base rapidly. This rapid increase has led to the development of an industry body and the emergence of an entire ecosystem to form a comprehensive mobile platform. With the rapid increase of user-base, both malware content authors and security researchers have become vividly interested in this ecosystem. Reports from several anti-virus vendors seem to conclude that Android and other mobile ecosystems are facing the threat of malware and that year-over-year the number of discovered malware strains seems to be increasing.

In this regard, we first conduct a systematic survey of the Android ecosystem. We cover the entire breadth of the Android ecosystem in our survey when compared to other surveys that are highly focused on apps or malware. In our survey, we define the Android ecosystem components and identify the role they play in ensuring a secure ecosystem. We identify the challenges faced by each of the entities in the ecosystem and explain them in great detail. Subsequently, we systematize work published to address these issues. During our analysis, we observed several areas of Android ecosystem that have not been solved. Based on our analysis, we identified futures venues of research.

One such area which we identified for future work is the issue of colluding apps. Colluding apps involve two or more applications establishing covert channels of communication to extract sensitive data from the phone. Some of these colluding channels exist due to design inconsistencies while re-purposing the Linux kernel for Android OS. The problem of colluding apps is non-trivial since these channels operate in different levels of the Android architecture. We address the issue of colluding

app by proposing a novel policy-based framework to monitor the colluding channels. We design USB-Droid, a light-weight framework for policy enforcement that operates on Binder transactions. By adding several new modules and enhancing existing modules, we realize the proposed system on Android 4.4.4 release 2. Our evaluation with Nexus 5 KitKat OS shows the ease of use of the system and the overhead of the system. With an average delay of less than 0.01 seconds, a boot-time impact of less than 8% and overall performance impact of less than 7%, our solution is indeed light-weight. Based on our evaluation with 125 apps, we notice that our system has the problem of false positives. However, since manual review is mandated by the system and since the user has the required context information, the problem of false positives is mitigated according to the privacy concerns of the user. We are conducting a feasibility study to automate the decision after flagging a suspicious transaction through covert channels.

REFERENCES

[1] Ben Elgin. Google Buys Android for Its Mobile Arsenal [Online], August 2005. Available: http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal

[2] Gareth Beavis. A complete history of Android [Online], September 2008. Available: http://www.techradar.com/us/news/phone-and-communications/mobile-phones/a-complete-history-of-android-470327

[3] Open Handset Alliance [Online] November 2007. Available: http://www.openhandsetalliance.com/press_110507.html

[4] Dan Graziano. Daily Android activations grow to 1.5 million, Google Play surpasses 50 billion downloads [Online] July 2013. Available: http://bgr.com/2013/07/20/android-activations-app-downloads/

[5] Technology Research — Gartner Inc. [Online] April 2015. Available: http://www.gartner.com/

[6] Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013 [Online] February 2014. Available: http://www.gartner.com/newsroom/id/2665715

[7] Number of available Android applications [Online] April 2015. Available: http://www.appbrain.com/stats/number-of-android-apps

[8] McAfee labs. McAfee threats report: Second quarter 2013. Technical Report, McAfee Labs, 2013 Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2013.pdf

[9] McAfee Labs Report Previews 2015 Developments in Exploits and Evasion [Online] December 2014. Available: http://www.mcafee.com/us/about/news/2014/q4/20141209-01.aspx

[10] Yajin Zhou and Xuxian Jiang. 2012. "Dissecting Android Malware: Characterization and Evolution." *In Proceedings of the 2012 IEEE Symposium on Security and Privacy* (SP 2012). IEEE Computer Society, Washington, DC, USA, 95-109.

[11] Claudio Marforio, Hubert Ritzdorf, Aurlien Francillon, and Srdjan Capkun. 2012. "Analysis of the communication between colluding applications on modern smartphones." *In Proceedings of the 28th Annual Computer Security Applications Conference* (ACSAC '12). ACM, New York, NY, USA, 51-60.

[12] Chandra, S., Lin, Z., Kundu, A., & Khan, L. (2014). Towards a Systematic Study of the Covert Channel Attacks in Smartphones. Technical Report, University of Texas at Dallas.

[13] Android Apps on Google Play [Online] April 2015. Available: https://play.google.com/store/apps?hl=en

[14] Amazon Appstore for Android [Online] April 2015. Available: http://www.amazon.com/mobile-apps/b?node=2350149011

[15] F-Droid [Online] April 2015. Available: https://f-droid.org/

[16] Baidu App Store [Online] April 2015. Available: http://as.baidu.com/

[17] Anzhi [Online] April 2015. Available: http://www.anzhi.com/

[18] Tencent's App Store [Online] April 2015. Available: http://android.myapp.com/

[19] Launch Checklist [Online] April 2015. Available: http://developer.android.com/distribute/googleplay/publish/preparing.html

[20] Android Open Source Project [Online] April 2015. Available: https://source.android.com/

[21] Introducing ART [Online] April 2015. Available: http://source.android.com/devices/tech/dalvik/art.html

[22] Dalvik Technical Information [Online] April 2015. Available: http://source.android.com/devices/tech/dalvik/index.html

[23] Android NDK [Online] April 2015: Available: http://developer.android.com/tools/sdk/ndk/index.html

[24] Jerome H. Saltzer. 1974. Protection and the control of information sharing in multics. *Commun. ACM* 17, 7 (July 1974), 388-402. DOI=10.1145/361011.361067 http://doi.acm.org/10.1145/361011.361067

[25] Chen, Q. A., Qian, Z., & Mao, Z. M. (2014, August). Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proc. 23rd USENIX Security Symposium (SEC14)*, USENIX Association.

[26] Lin, C. C., Li, H., Zhou, X., & Wang, X. (2014, February). Screenmilker: How to milk your android screen for secrets. *In Proceedings of 21th USENIX Network Distributed System Security Symposium*, 2014.

[27] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, Klara Nahrstedt. "Identity, location, disease and more: inferring your secrets from android public resources" *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (CCS 2013)

[28] Schlegel, Roman, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. "Soundcomber: A Stealthy and Context-Aware Sound

Trojan for Smartphones." *In Proceedings of the 21st Annual Network and Distributed System Security Symposium* (NDSS'11) pp. 17-33. 2011.

[29] NCSC, NSA, "Covert Channel Analysis of Trusted Systems." NSA/NCSC Rainbow Series publications, 1993.

[30] Android and Security [Online] February 2012. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html

[31] Dashboards — Android [Online] April 2015. Available: https://developer.android.com/about/dashboards/index.html

[32] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. 2013. "Vetting undesirable behaviors in android apps with permission use analysis," *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (CCS '13). ACM, New York, NY, USA, 611-622.

[33] David Sounthiraraj and Justin Sahs and Zhiqiang Lin and Latifur Khan and Garrett Greenwood, "SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps," *In Proceedings of the 21st Annual Network and Distributed System Security Symposium* (NDSS'14). San Diego, CA, USA.

[34] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgrtner, Bernd Freisleben, and Matthew Smith. 2012, "Why eve and mallory love Android: an analysis of Android SSL (in)security," *In Proceedings of the 2012 ACM Conference on Computer and Communications Security* (CCS '12). ACM, New York, NY, USA, 50-61.

[35] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013, "Rethinking SSL development in an appified world," *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (CCS '13). ACM, New York, NY, USA, 49-60.

[36] M. Marlinspike. New tricks for defeating SSL in practice. *In BlackHat DC*, February 2009.

[37] Shekhar, Shashi, Michael Dietz, and Dan S. Wallach. "AdSplit: separating smartphone advertising from applications." *In Proceedings of the 21st USENIX Conference on Security Symposium*, pp. 28-28. USENIX Association, 2012.

[38] Franziska Roesner and Tadayoshi Kohno. 2013. "Securing embedded user interfaces: Android and beyond." *In Proceedings of the 22nd USENIX Conference on Security* (SEC'13). USENIX Association, Berkeley, CA, USA, 97-112.

[39] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. "Brahmastra: driving apps to test the security of third-party components." *In Proceedings of the 23rd USENIX Conference on Security Symposium* (SEC'14). USENIX Association, Berkeley, CA, USA, 1021-1036.

[40] Octeau, Damien, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis." *In Proceedings of the 22nd USENIX Security Symposium*. 2013.

[41] Wu, C., Zhou, Y., Patel, K., Liang, Z., & Jiang, X. (2014, February). AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*.

[42] Sun, M., & Tan, G. (2014, July). NativeGuard: protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (pp. 165-176)*. ACM.

[43] Portokalidis, Georgios, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. "Paranoid Android: versatile protection for smartphones." *In Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 347-356. ACM, 2010.

[44] Reina, A., Fattori, A., & Cavallaro, L. (2013). A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec 2014*, April.

[45] Yang, C., Xu, Z., & Gu, G. (2014). Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. *ESORICS 2014*.

[46] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. "Quire: lightweight provenance for smart phone operating systems." *In Proceedings of the 20th USENIX Conference on Security* (SEC'11).

[47] Backes, M., Bugiel, S., & Gerling, S. (2014). Scippa: system-centric IPC provenance on Android. *Proceedings of the 30th Annual Computer Security Applications Conference 2014*.

[48] Enck, W., Gilbert, P., Chun, B. G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2014). TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3), 99-106.

[49] Anthony Desnos and Patrik Lantz. DroidBox: An Android Application Sandbox for Dynamic Analysis. [Online] August 2011. Available: http://project.honeynet.org/gsoc2011/slot5

[50] Anubis - Malware Analysis for Unknown Binaries [Online] April 2015. Available: https://anubis.iseclab.org/

[51] Arzt, Steven, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. (2014, June). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (p. 29)*. ACM.

[52] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. "On lightweight mobile phone application certification." *In Proceedings of the 16th ACM Conference on Computer and Communications Security* (CCS '09).

[53] Chen, Kevin Zhijie, Noah Johnson, Vijay DSilva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward Wu, Martin Rinard, and Dawn Song. "Contextual Policy Enforcement in Android Applications with Permission Event Graphs." *In Proceedings of the Network and Distributed System Security Symposium*, 2013.

[54] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. 2013. "Unauthorized origin crossing on mobile platforms: threats and mitigation." *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (CCS '13). ACM, New York, NY, USA, 635-646.

[55] Zhang, M., Duan, Y., Yin, H.,& Zhao, Z. (2014, November). Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency

Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (pp. 1105-1116)*. ACM.

[56] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. 2009. "Semantically Rich Application-Centric Security in Android." *In Proceedings of the 2009 Annual Computer Security Applications Conference* (ACSAC '09).

[57] Rubin Xu, Hassen Sadi, and Ross Anderson. 2012. "Aurasium: practical policy enforcement for Android applications." *In Proceedings of the 21st USENIX Conference on Security Symposium* (Security'12). USENIX Association, Berkeley, CA, USA, 27-27.

[58] Ongtang, Machigar, Kevin Butler, and Patrick McDaniel. "Porscha: Policy oriented secure content handling in Android." *In Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 221-230. ACM, 2010.

[59] Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012, February). Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (pp. 317-326)*. ACM.

[60] Crussell, Jonathan, Clint Gibler, and Hao Chen. "Attack of the clones: Detecting cloned applications on android markets." *In Computer Security ESORICS 2012*, pp. 37-54. Springer Berlin Heidelberg, 2012.

[61] Jonathan Crussell, Clint Gibler, Hao Chen: "AnDarwin: Scalable Detection of Semantically Similar Android Applications." *ESORICS 2013*: 182-199

[62] Armando, A., Merlo, A., Migliardi, M., & Verderame, L. (2012). Would you mind forking this process? A denial of service attack on Android (and some

countermeasures). In *Information Security and Privacy Research (pp. 13-24)*. Springer Berlin Heidelberg.

[63] CVE - CVE-2014-3153 [Online] April 2015. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153

[64] Cybercriminals Improve Android Malware Stealth Routines with OBAD [Online] June 2013. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/cybercriminals-improve-android-malware-stealth-routines-with-obad/

[65] Detecting Hidden Administrator Apps on Your Mobile Device [Online] June 2013. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/detecting-hidden-administrator-apps-on-your-device/

[66] CVE - CVE-2011-3874 [Online] April 2015. Available: http://www.cvedetails.com/cve/CVE-2011-3874/

[67] CVE-CVE-2011-1823 [Online] April 2015. Available: http://www.cvedetails.com/cve/CVE-2011-1823/

[68] CVE-CVE-2011-1149 [Online] April 2015. Available: http://www.cvedetails.com/cve/CVE-2011-1149/

[69] Davi, Lucas, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nrnberger, and Ahmad-Reza Sadeghi. "MoCFI: A framework to mitigate control-flow attacks on smartphones." *In Symposium on Network and Distributed System Security* (NDSS). 2012.

[70] CVE-CVE-2010-1807 [Online] April 2015. Available: http://www.cvedetails.com/cve/CVE-2010-1807/

[71] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. 2013, "Predictability of Android OpenSSL's pseudo random number generator," *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (CCS '13). ACM, New York, NY, USA, 659-668.

[72] Building Web Apps in WebView [Online] April 2015. Available: http://developer.android.com/guide/webapps/webview.html

[73] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. "Attacks on WebView in the Android system." *In Proceedings of the 27th Annual Computer Security Applications Conference* (ACSAC '11). ACM, New York, NY, USA, 343-352.

[74] Georgiev, Martin, Suman Jana, and Vitaly Shmatikov. "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks," *Network and Distributed System Security Symposium* (2014).

[75] Jin, X., Hu, X., Ying, K., Du, W., Yin, H., & Peri, G. N. (2014, November). Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (pp. 66-77)*. ACM.

[76] Yajin Zhou, Xuxian Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications" *Proceedings of the 20th Network and Distributed System Security Symposium* (NDSS 2013) San Diego, CA, February 2013.

[77] Xing, Luyi; Pan, Xiaorui; Wang, Rui; Yuan, Kan; Wang, XiaoFeng, "Upgrading Your Android, Elevating My Malware: Privilege Escalation through Mobile OS Updating," *Security and Privacy (SP), 2014 IEEE Symposium on* , pp.393,408, 18-21 May 2014

[78] Android Fake ID Vulnerability Lets Malware Impersonate Trusted Applications, Puts All Android Users Since January 2010 At Risk [Online] July 2014. Available: https://bluebox.com/technical/android-fake-id-vulnerability/

[79] CVE - CVE-2013-4787 [Online] April 2015. Available: http://www.cvedetails.com/cve/CVE-2013-4787/

[80] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., & Vigna, G. (2014, February). Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS (Vol. 14, pp. 23-26)*.

[81] Zhaohui Wang and Angelos Stavrou. 2010. "Exploiting smart-phone USB connectivity for fun and profit." *In Proceedings of the 26th Annual Computer Security Applications Conference* (ACSAC '10). ACM, New York, NY, USA, 357-366.

[82] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. 2012. "Practicality of accelerometer side channels on smartphones." *In Proceedings of the 28th Annual Computer Security Applications Conference* (ACSAC '12). ACM, New York, NY, USA, 41-50.

[83] Peng, Hao, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. "Using probabilistic generative models for ranking risks of android apps." *In Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 241-252. ACM, 2012.

[84] Pandita Rahul, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. "WHYPER: towards automating risk assessment of mobile applications." *In Proceedings of the 22nd USENIX Security Symposium*, Washington DC, USA, pp. 14-16. 2013.

[85] Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., & Chen, Z. (2014, November). AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1354-1365). ACM.

[86] Gorla, A., Tavecchia, I., Gross, F., & Zeller, A. (2014, May). Checking app behavior against app descriptions. In *ICSE* (pp. 1025-1035).

[87] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. "PScout: analyzing the Android permission specification." *In Proceedings of the 2012 ACM Conference on Computer and Communications Security* (CCS '12). ACM, New York, NY, USA, 217-228.

[88] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang and Binyu Zang. "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis." *In Proc. of the 20th ACM Conference on Computer and Communications Security* (CCS'13), Berlin, Germany, November 2013.

[89] Fedler, R., Kulicke, M., & Schtte, J. (2013, November). Native code execution control for attack mitigation on android. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (pp. 15-20). ACM.

[90] Ho, T. H., Dean, D., Gu, X., & Enck, W. (2014, March). PREC: practical root exploit containment for android devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy* (pp. 187-198). ACM.

[91] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. "CHEX: statically vetting Android apps for component hijacking vulnerabilities." *In Pro-*

ceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12). ACM, New York, NY, USA, 229-240.

[92] Grace, Michael, 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.

[93] Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards taming privilege-escalation attacks on Android." *In Proceedings of the 19th Annual Symposium on Network and Distributed System Security.* 2012.

[94] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. "Permission re-delegation: attacks and defenses." *In Proceedings of the 20th USENIX Conference on Security* (SEC'11). USENIX Association, Berkeley, CA, USA.

[95] Shared Preferences [Online] April 2015. Available: http://developer.android.com/reference/android/content/SharedPreferences.html

[96] logcat [Online] April 2015. Available: http://developer.android.com/tools/help/logcat.html

[97] Adwait Nadkarni and William Enck. 2013. "Preventing accidental data disclosure in modern operating systems." *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (CCS '13). ACM, New York, NY, USA, 1029-1042.

[98] Zhang, Mu, and Heng Yin. "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Ap-

plications." *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium*. 2014.

[99] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. "AppIntent: analyzing sensitive data transmission in android for privacy leakage detection." *In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (CCS '13). ACM, New York, NY, USA, 1043-1054.

[100] Smalley, Stephen, and Robert Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android." *In Network & Distributed System Security Symposium* (NDSS '13). 2013.

[101] Bugiel, Sven, Stephan Heuser, and Ahmad-Reza Sadeghi. "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies." *In Proceedings of 22nd USENIX Security Symposium* (USENIX Security'13). 2013.

[102] Bugiel, S., Heuser, S., & Sadeghi, A. R. (2013, August). Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *USENIX Security* (pp. 131-146).

[103] Heuser, S., Nadkarni, A., Enck, W., & Sadeghi, A. R. (2014, March). Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium* (SEC'14).