The Pennsylvania State University
The Graduate School

ANALYSIS TECHNIQUES FOR MOBILE OPERATING SYSTEM SECURITY

A Dissertation in
Computer Science and Engineering
by
William Harold Enck

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2011

The dissertation of William Harold Enck was reviewed and approved* by the following:

Patrick D. McDaniel
Associate Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Trent R. Jaeger
Associate Professor of Computer Science and Engineering

Thomas F. La Porta
Distinguished Professor of Computer Science and Engineering

Eileen Kane
Professor of Law

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Devices such as smartphones running mobile operating systems have become an integral part of society. Current smartphones are a response to the Internet's influence on computing technology: devices provide nearly pervasive access to information and commoditize a seemingly endless number of services. However, smartphones are more than ultra-portable Web browsers. They combine the expanse knowledge and information available on the Internet with local context made accessible through hardware features such as GPS receivers, microphones, cameras, and accelerometers. In the past several years, smartphone innovation and popularity has surged in response to more open programming interfaces and network capabilities. Underlying this valuable innovation lies increased security risk for users and providers of content and cellular service.

In this dissertation, we explore the limitations of existing mobile operating systems to protect end users from undesirable behavior by downloaded applications. Existing security frameworks define security policy in terms of permissions. We use requested permissions to focuses security analysis of available applications. First, we consider which permissions applications request and show that this limited information can prevent applications with dangerous functionality from being installed. Second, we consider what applications do with permissions. We design and build a framework for realtime dynamic taint analysis to identify misuse of information such as location and phone identifiers. Finally, we consider what applications can do with permissions based on implemented functionality. In doing so, we use several types of source code analysis to identify both dangerous behavior and vulnerabilities in decompiled applications. While we find the coarseness of permissions to be insufficient in several cases, the permission-based model fundamentally aided our analysis, demonstrating new potential for protecting future mobile platforms.

Table of Contents

List of Figures

List of Tables

Acknowledgments

A dissertation is a reflection if its author, and as the saying goes, “we are a product of our environment.” Therefore, this dissertation is in large part a result of external influence. The completion of my dissertation marks the end of eleven years of study at The Pennsylvania State University, earning a Bachelors, a Masters, and now a PhD degree. While this particularly long student tenure has been the focus of much lighthearted harassment at family gatherings, Penn State has defined much of who I am, as well as what I am yet to become.

While I have stayed in one place, the world around me has changed. In my first semester of undergrad, I was fortunate to meet Jason Schwier, Tom Richardson, and Alex Rovnan. It is rare that one meets friends with similar drive, passion, and dedication for a related subject area. Over the next four years, we took many similar classes, studied endless hours together, and supported one another through the personal chapters of our lives. I give great thanks to their friendship and support that laid the foundation of my academic career. I would also be remiss not to mention the Penn State Linux Users Group (PSULUG) and its members between 2001 and 2004. Having served as president and secretary of this eccentric group, I must accredit much of my knowledge of systems and debugging to countless hours fiddling with the more esoteric details of UNIX. The members of PSULUG helped drive my enthusiasm and understanding for low level computing systems.

Much of who I am has been defined by my tenure in graduate school. I chose to stay at Penn State for my Masters mostly out rational thought. Almost everyone I knew in my field either went for a Masters degree directly after undergrad, or juggled work and classes to earn a Masters degree while working full-time. Furthermore, I wasn’t passionate enough for my undergraduate area of focus, microarchitecture. I knew earning a PhD was possible, but I could not foresee a life studying the area.

My passion for research was ignited during my first semester of graduate school. Searching for classes to take, I noticed a new class on computer security; however, it was full. Having always had a particular interest in security when dabbling with UNIX systems, I sought the aid of the CSE administrative staff, who knew me as a former top undergraduate student. By their grace, I was able to schedule the class. On the first day of class, I met my advisor, Patrick McDaniel. Consequently, this was also Patrick’s

first day teaching at Penn State. I could go on at length describing my experiences in the class and the years that followed under this tutelage. However, I think it is best put succinctly. Apart from my family, Patrick has had the single most influential impact on my life. His passion and spirit are the reason I chose to pursue a PhD, the reason I am passionate for research, and the reason I am choosing an academic career. He has helped me grow in ways words cannot describe, and similarly words cannot describe my gratitude. Along with Patrick, I must thank his wife, Megan, and his children, Sinclair and Emerson, for all of their sacrifices while Patrick started a research lab. I also thank Megan, who frequently has been called "lab mom," for all that she has done, from providing the hospitality of her home during paper deadlines, to being Patrick's bedrock of support.

Close on Patrick's heels in terms of impact in my life is Patrick Traynor, or "P2" as we fondly refer to him in order to disambiguate our environment overrun with "Patricks." P2 helped me begin my research career as we discovered vulnerabilities in cellular networks. Some of my fondest memories of graduate school occurred at 2am, discussing the project while keeping ourselves awake and sane playing baseball with a paper ball wrapped in masking tape and poster tube as a bat. From this (by which I mean the research collaboration, and not the baseball), I learned the power of collaboration and its ability to inspire the creative process. Since then, P2 has been many things, including: mentor, roommate, friend, and brother. Few are lucky to find such a person; even fewer find two. Kevin Butler is a former lab-mate and roommate of five years. We have shared so many adventures, and so many conversations of both research and personal nature, in all hours of the day and night, that "friend" does not even begin to articulate our relationship. He has impacted much of who I am today. Together with Patrick McDaniel and Patrick Traynor, Kevin is part of my family.

The Systems and Internet and Infrastructure Security Laboratory, or SIIS Lab, is what brought P1, P2, Kevin, and myself together. Yet unmentioned is Boniface Hicks, the final member of our founding group. I consider myself blessed to have had the opportunity to sit in a desk beside his. Boniface is a remarkable person, who consequently studied a complementary area of security. We exchanged many ideas and insights into systems and programming languages. Much of what I know about information flow logics can be attributed to his explanations. I am the last of the founding members of the SIIS Lab to graduate, and I have had the privilege of working with many students earning Masters and PhD degrees. In particular, I would like to make note of Josh Schiffman, who is always eager to discuss systems architectures; Steve McLaughlin, who has taught me much of the philosophical side of research; and Sandra Rueda, who often ensured that the jovial side of running the lab was not overlooked. I also thank Machigar Ongtang and Damien Octeau for their direct contributions to my research.

I would also like to thank the members of my dissertation committee. I have worked with Tom La Porta on many occasions. During my first semester of graduate school, Tom taught me mobile telecommunications networks. He is also a co-author on the telecommunications security research that helped inspire me as a researcher. Over the years, Tom has been an invaluable resource for both research and pursuing an academic career. Trent Jaeger has taught me much of what I know of the foundational concepts of operating systems security. I continually forget that we have published relatively few papers together, because we continually converse about research. I cannot thank him enough for all of his help and insights. Finally, Eileen Kane has been a valuable addition to my otherwise technically oriented committee. Her legal perspective has helped ground the privacy aspects of my dissertation.

Outside of Penn State, I would like to thank Jaeyeon Jung for inviting me to do a summer internship with Intel Labs in Seattle. The combination of our interests and expertise lead to the creation of TaintDroid, one of the foundational pieces of my dissertation. This internship led to many untold rewards and opportunities.

In addition to my collaborators, I wish to thank the many unnamed graduate students I have met while at Penn State. They have provided friendship and countless healthy distractions from research that have helped maintain my sanity. Graduate school is a long and grueling process, and the support of one another does not go unnoticed.

Finally, I give my deepest thanks to my family for all of their support and inspiration. My sisters, Mel Enck and Robin Enck Medic, have never set the bar low, whether it be teachers' expectations in elementary school, or earning their Veterinary and Medical Doctorates, respectively. My late grandparents, William J. and Earla Enck, have also been a foundational inspiration. My grandmother was one of the most compassionate people I've met, and I will never forget my grandfather's words, "where there's a will, there's a way, and we've got two." Lastly, my mother, Clarissa Enck, and my father, John Enck, have guided me through good times and bad, and have taught me what makes this life special. They have opened for me a world of opportunity, and it is to them that this dissertation is dedicated.

Dedication

To my mother and father, for opening a world of opportunity.

Chapter 1

Introduction

The first full-fledged mobile operating system with networking capabilities was designed for a phone. These so called "smartphones" appeared in the early 2000's. While the transition to today's definition of smartphone was gradual, the most notable early mobile platform is the Symbian OS, which gained popularity in Europe and Asia, but not North America. In contrast, the early North American smartphone market was dominated by RIM's BlackBerry handset tailored for business users. However, the platform's vertical focus on pervasive access to email and personal information management (PIM) had limited appeal to the general populous. It was not until Apple's release of the iPhone 3G and App Store in 2008 that smartphone popularity in North America began to surge.

A smartphone is classically defined as a mobile phone that allows the user to download and run third-party applications from the Internet. Contrasted with feature phones, which provide enhanced functionality fixed by the device manufacturer or service provider, smartphones enable the user to decide how to extend a phone's functionally based on available applications. However, as demonstrated by the slow adoption of smartphones in North America during most of the 2000's, accessibility of applications plays an important role in both the use of applications on smartphones and innovation in the domain.

An *application market* provides a central point for application distribution and discovery. Markets such as Apple's App Store and Google's Android Market remove barriers of entry for developers by simplifying sales and distribution. Combined with these platforms' relatively easy to use application programming interfaces (APIs), the markets are lush with hundreds of thousands of applications. On the consumer front, markets simplify discovery, purchase, and installation of applications. This process is self-contained on the phone handset. Users search for applications matching keyword criteria using an on-phone market user interface. After choosing an application from the search results, the user purchases it (if necessary), and the application is downloaded over a wireless interface and installed. This simplicity on both developer and consumer fronts creates an incubator for innovation.

Smartphones offer a unique environment for innovation. In many regards, smartphones are the logical conclusion the Internet's influence of technology over the last decade: they offer access to information from ostensibly anywhere; they allow a diverse and seemingly limitless set of services; and they commoditize those services into simple to use interfaces for consumers. In fact, smartphones share many characteristics with Web browsers, possibly more so than with traditional operating systems. However, a smartphone is more than an ultra-portable Web browser. They combine the expanse knowledge and information available on the Internet with local context made accessible through hardware features such as GPS receivers, microphones, cameras, and accelerometers. The result is innovation that previously only appeared in science fiction. Take for example, an augmented reality application that visually translates signs on restaurants and stores for a tourist visiting an area that speaks a foreign language [1]. Through such innovations, smartphones have become invaluable resources and are reshaping the direction of computing technology.

Unfortunately, this innovation is not without security risk. Smartphone applications continually gather information as a matter of operation. For example, a weather forecast application uses the phone's geographic coordinates to report current and future weather conditions. However, collected information extends beyond simple environmental context from hardware sensors. It also includes the user's social interactions with the physical world, for example, preferences when searching for nearby restaurants, and queries for public transit schedules. All of this information is stored on a small and easy to lose device that always travels with the user, drifts seamlessly between "unknown" wireless networks, and runs applications from largely unknown developers.

The source of applications is a particularly interesting security risk. The simplicity of platform APIs and market interfaces creates a low barrier of entry for developers. The resulting cottage industry is filled with largely unknown developers that do not always fully understand the security and privacy requirements of the environment. In particular, as is discussed in later chapters, developers frequently do not acquire *informed consent* before using private information in ways that may negatively impact the user.

1.1 Thesis Statement

It is a common misconception that application markets can completely eliminate malicious and dangerous applications before distribution [2]. The central point of application management does provide a very valuable feature: if a malicious application is detected, not only can its distribution be stunted, but so called “kill switches” allow markets to remove malicious applications from deployed phone handsets [3]. However, markets are limited in the security guarantees that they can provide. First, security requirements are often individualized to specific handsets. Each user has a set of expectations about what an application should and should not do. Hence, the definition of security cannot be articulated coherently at this level. Second, even with an acceptable definition of security, developers submit thousands of applications each month. Applying anything more than simple analysis to each application is logistically impractical, if not impossible.

The primary source of smartphone security is on the handset. This security is frequently defined in terms of *permissions*. Platform developers appropriately assume a phone handset has one physical user, thereby making applications first-class principals in the security policy. Each application is assigned a set of permissions, which are capabilities describing the information and resources the application may access. Application permissions are granted by the user, effectively moving security decisions from the market to the handset, where a more appropriate level of context is available.

The focus of this dissertation is to understand the limitations of existing mobile smartphone operating system security frameworks, and to suggest techniques to improve the state of the art. More specifically, we hope to identify the most prominent risks faced by users and determine if failures result from protection mechanism limitations. To do this, we study the behavior of popular applications.

As an artifact of our studies, we observe that most smartphone applications have a vertical purpose. We surmise this characteristic results from three driving influences: *a*) smartphones are an evolutionary response of Internet communication, effectively commoditizing services; *b*) the cottage market driving smartphone applications forces developers to realize conceptual ideas quickly; and *c*) the limited user interfaces of handsets restricts application complexity.

The vertical purpose nature of smartphone applications has two apparent side effects: 1) the user’s expectations of application behavior are often clear; and 2) applications frequently request a limited number of permissions. Note that despite initial fears, we have not found applications to be simply requesting all permissions. While it is impossible to determine if this is a result of honest developers or fear of bad reputation, the existence of this phenomenon has been corroborated by related studies [4].

We study the intersection of permission requests and application behavior using three analysis techniques. First, we use static analysis of configuration to understand *what permissions applications request.* These permissions represent an upper bound on the functionality an application may exercise at runtime. While application behavior permits successful articulation of security goals at this granularity, we identified several cases where this permission-defined protection domain is too coarse. Second, we build a framework to perform dynamic taint analysis of privacy sensitive information to understand *what applications do with permissions.* This analysis effectively allows us to "look inside" of applications to identify when sensitive information is unexpectantly shared with network servers. However, dynamic taint analysis only permits observation of a rather vertical (albeit important) type of dangerous behavior. It is also limited to excised functionality and does not identify which part of the application code was responsible (e.g., an included library vs. the main code). Third and finally we use static source code analysis to understand *what applications can do with permissions based on implemented functionality.* Here, we look for not only information misuse, but also a broader class of dangerous functionality and vulnerabilities.

We observe that requested permissions have become a valuable means of assessing risk and informing analysis. The discoveries and observations made during our studies of smartphone applications inform the following thesis statement.

> *Permission requests by smartphone applications can be used to focus security analysis to practically identify dangerous behavior and vulnerabilities.*

Evidence of this thesis can be found in each of our studies. Our first study identifies dangerous behavior directly from permissions. The vertical purpose nature of smartphone applications and the corresponding permission requests allows us to practically define security invariants that do not prevent permissions combinations required by many applications. Our second study uses permissions to determine which applications to investigate with dynamic taint analysis. Our final study uses knowledge of permission semantics to identify which programming interfaces should be analyzed for dangerous behavior and vulnerabilities, and how the interfaces should and should not be used.

Our study results indicate future directions for smartphone security solutions. First, we observe the need to mitigate misuse of privacy sensitive values. We find the existing permission-based protection mechanism insufficient in its current application. We suggest an improvement based on careful privilege separation in our concluding remarks.

Second, we observe that no granularity of permission assignment will remove all security risks, and that the security analysis techniques describe herein should be applied as the basis for an ongoing evaluation of the security hygiene of application markets. In some cases, it may be practical to apply the analysis before application distribution; however, to address the diverse security requirements of smartphone users, we see the need for continual evaluation and rating of applications after they have appeared in markets.

Finally, we note that this dissertation focuses on abuse of information and resources by applications as it affects end users. There are many other important areas relating to smartphone security. In particular, we make no effort to address physical attacks on smartphones, attacks on telecommunications infrastructure, enterprise security concerns, or digital rights management.

1.2 Contributions

In this dissertation, we make the following contributions:

- *We propose a model of lightweight certification based purely upon application configuration data.* Each end user has different security requirements. The Kirin [5] enhancement to Android's application installer certifies an application's configuration policy, including the permissions it requests, based on a local criteria specification. The certification criteria are security invariants describing dangerous application configurations. If an application fails to meet the criteria, it is not installed. We define nine valuable invariants based on security requirements engineering applied to the Android operating system to demonstrate the usefulness of the limited policy language. We evaluate the invariants against 311 popular applications and find a low number of false positives.

- *We design and implement a framework for monitoring privacy sensitive information on smartphones and use it to identify risks in popular applications.* In some cases, configuration-level analysis is too coarse to distinguish between dangerous and benign functionality in applications. The TaintDroid [6] augmentation of the Android platform provides dynamic taint analysis for privacy sensitive information such as location, microphone, camera, and phone identifiers. We make careful trade-offs on tracking granularity to achieve realtime monitoring performance. While TaintDroid is specific to Android, its architecture is portable to other VM interpreter based platforms (e.g., Java-based phones such as BlackBerry). We use

TaintDroid to analyze 30 popular applications and identify 15 applications sharing location information with advertisement servers, and 7 applications sharing phone identifiers with remote servers without user knowledge.

- *We provide a preliminary characterization of security in Android applications using source code analysis.* Dynamic taint analysis is limited to detecting dangerous functionality based on information flows. Furthermore, only exercised functionality is analyzed. Using the `ded` decompiler, we recover over 21 million lines of source code for source code the top 1,100 popular free applications in the Android Market. This study [7] defines source code analysis specifications for both dangerous functionality and vulnerabilities. Manual inspection of analysis results finds minimal existence of overtly dangerous or readily exploitable applications. The primary source of identified risk in the studied applications is the use of phone identifiers and location by developers.

- *We identify that existing smartphone OS security frameworks lack sufficient* mediation *for a clear and present risk for end users.* More specifically, they lack sufficient control of privacy sensitive information such as location and phone identifiers. At the crux of this limitation is a conflict between operational and security requirements that is specific to the smartphone environment. To benefit from the innovation in the smartphone application domain, users must give applications access to both privacy sensitive information such as location *and* the Internet. However, once information enters a process, the OS cannot mediate its disclosure to network servers.

1.3 Dissertation Outline

The goal of this dissertation is to understand limitations of existing smartphone security frameworks by studying the functional requirements and operation of deployed applications. In order to achieve this objective, we concert our focus on the Android platform, which is both representative of current and future computing trends in the smartphone domain, and provides the necessary openness of documentation and platform source code to facilitate study, analysis, and experimentation.

In Chapter 2, we describe the current state of security in smartphones and similar mobile operating systems. We begin with an overview of existing and expected threats by malware. Next, we discuss application markets in detail, enumerating their limitations and potential with respect to security. Chapter 2 concludes with detailed information of the Android platform and its security framework. This background information is followed by a discussion of foundational concepts and related work in Chapter 3.

In Chapter 4, we present an analysis approach considering static application configuration. Driving this investigation is the Kirin enhancement to Android's application installer. We present formal logic for analyzing applications based on their configuration and describe a security requirements engineering technique to articulate security goals limited by a single application's configuration policy.

In Chapter 5, we extend analysis to fine-grained runtime information tracking. The static configuration analysis in Chapter 4 proves problematic when security policy allows an application to access both sensitive information and the Internet. We present the TaintDroid realtime analysis framework to look inside of applications and understand how sensitive information is used. Using this dynamic taint analysis, we study popular applications and identify misuse of privacy sensitive values such as location and phone identifiers.

In Chapter 6, we revert to static analysis, but do so with knowledge of application source code obtained using the `ded` decompiler. Analysis of source code allows us to look for more than misuse of values, but also control flows that lead to dangerous functionality. Furthermore, we use source code analysis to identify vulnerabilities in applications. The presented analysis studies over 21 million lines of code across 1,100 popular applications demonstrates the value of transparency for security. Here, we identify information misuse of location and phone identifiers as the most common risk for end users.

Finally, Chapter 7 concludes our discussion of the security limitations of mobile operating systems and discusses directions for future work.

Chapter 2

Mobile Operating System Security

Smartphone platforms are the canonical example of mobile operating systems. For the purposes of this dissertation, we consider a mobile operating system to one for a portable network-capable device such as a phone, tablet, or netbook. While desktop operating systems are sometimes run on tablets and netbooks (or other small laptops), our focus is on resource constrained devices and the limitations they impose.

This chapter provides background on smartphone security necessary to understand the remaining chapters. While the discussion focuses on smartphones, many explanations and observations apply to portable devices running variants of operating systems designed for smartphones. We begin by discussing threats to smartphones. We then describe the benefits and limitations of application markets. Finally, the chapter concludes with a case study of the Android platform and its security framework.

2.1 Smartphone Threats

In this section, we discuss threats to smartphones. We begin with an overview of existing malware and then classify types of malware that is emerging or is expected to occur. Finally, we discuss privacy concerns and how not all dangerous functionality can be characterized explicitly as malware.

2.1.1 Malware

The first smartphone virus was observed in 2004. While Cabir [8] carries a benign payload, it demonstrated the effectiveness of Bluetooth as a propagation vector. Its most notable outbreak was at the 2005 World Championships in Athletics [9]. More interestingly, Cabir did not exploit any vulnerabilities. It operated entirely within the security parameters of both its infected host (Symbian OS) and Bluetooth. Instead, it leveraged flaws in the user interface. While a victim is in range, Cabir continually sends file transfer requests. When the user chooses "no," another request promptly appears, frustrating the user who subsequently answers "yes" repeatedly in an effort to use the phone [10].

Cabir was followed by a series of viruses and Trojans targeting the Symbian Series 60 platform, each increasing in complexity and features. Based on Cabir, Lasco [11] additionally infects all available software package (SIS) files residing on the phone on the assumption that the user might share them. Commwarrior [12] added MMS propagation in addition to Bluetooth. Early variants of Commwarrior attempt to replicate via Bluetooth between 8am and midnight (when the user is mobile) and via MMS between midnight and 7am (when the user will not see error messages resulting from sending an MMS to non-mobile devices). Originally masquerading as a theme manager, the Skulls [13] Trojan provided one of the first destructive payloads. When installed, Skulls writes non-functioning versions of all applications to the `c:` drive, overriding identically named files in the firmware ROM `z:` drive. All applications are rendered useless and their icons are replaced with a skull and crossbones. Other Trojans, e.g., Drever [14], fight back by disabling Antivirus software. The Cardblock [15] Trojan embeds itself within a pirated copy of InstantSis (a utility to extract SIS software packages from a phone). However, Cardblock sets a random password on the phone's removable memory card, making the user's data inaccessible. While malware for other early smartphone operating systems such as Windows Mobile also appeared, smartphone malware little new malware was discovered after 2006 until recently [16].

The Android and iPhone platforms have also observed malware, albeit primarily "proof-of-concept." Oberheide [17] developed RootStrap as a rootkit delivery mechanism to demonstrate how an attacker might take advantage of zero-day vulnerabilities in the Linux kernel. Oberheide also developed malicious levels for a popular game (i.e., Angry Birds). This proof-of-concept malware misuses Android's account manager to obtain an Android Market authentication token an in turn installs applications without the user's knowledge [18]. Unfortunately, insufficient details of the vulnerability were available at the time of writing. Not all Android malware has been benign: a Trojan masquerading as a banking application attempted to steal user credentials [19]; and FakePlayer [20] sends SMS messages to premium rate numbers. In contrast to Android, to the best of our knowledge, iPhone malware has only affected jailbroken handsets. In the case of the Ikee.B worm [21], handsets with a default SSH password were infected with a botnet client. However, many cellular providers use private IP addresses for mobile devices, and the infection was limited to several countries.

To date, most phone malware has been either proof-of-concept or destructive, a characteristic often noted as resembling early PC malware. Recent PC malware more commonly scavenges for valuable information (e.g., passwords, address books) or joins a botnet [22]. The latter frequently enables denial of service (DoS)-based extortion. It is strongly believed that smartphone malware will move in similar directions [23, 10]. In fact, Pbstealer [24] already sends a user's address book to nearby Bluetooth devices, and Viver [25] and FakePlayer [20] send SMS messages to premium-rate numbers, providing the malware writer with direct monetary income.

Mobile phone literature has categorized phone malware from different perspectives. Guo et al. [26] consider categories of resulting network attacks. Cheng et al. [27] derive models based on infection vector (e.g., Bluetooth vs. MMS). However, we find a taxonomy based on an attacker's motivations [23] to be the most useful when discussion security from an OS perspective. We foresee the following motivations seeding future malware (the list is not intended to be exhaustive):

- *Proof-of-concept*: Such malware often emerges as new infection vectors are explored by malware writers and frequently have unintended consequences. For example, Cabir demonstrated Bluetooth-based distribution and inadvertently drained device batteries. As newer platforms such as iPhone and Android mature, we will continue to see proof-of-concept malware such as RootStrap [17].
- *Destructive*: Malware such as Skulls and Cardblock (described above) were designed with destructive motivations. While we believe malware with monetary incentives will overtake destructive malware, it will continue for the time being. Future malware may infect more than just the integrity of the phone. Current phone operating systems and applications heavily depend on cloud computing for storage and reliable backup. If malware, for example, deletes entries from the phone's address book, the data loss will propagate on the next cloud synchronization and subsequently affect all of the user's computing devices.
- *Premeditated spyware*: FlexiSPY (`www.flexispy.com`) is marketed as a tool to "catch cheating spouses" and is available for Symbian, Windows Mobile, BlackBerry, Android, and jailbroken iPhones. It provides location tracking, and remote listening. While malware variants exist, the software itself exhibits malware-like behavior and will likely be used for industrial espionage, amongst other purposes.

Such malware may be downloaded and installed directly by the adversary, e.g., when the user leaves the phone on a table.

- *Direct payoff*: Viver and FakePlayer (described above) directly compensate the malwares' author by sending messages to premium SMS numbers, and Terdial [28] makes voice calls to the South Pole. Such attacks impact both the end-user and the provider. Customers will contest the additional fees, leaving the provider with the expense. Any mechanism providing direct payment to a third party is a potential attack vector. For example, the iPhone platform has in-application content sales [29].
- *Information scavengers*: Web-based malware currently scours PCs for valuable address books and login credentials (e.g., usernames, passwords, and cookies for two-factor authentication for bank websites) [22]. Mobile phones are much more organized then their PC counterparts, making them better targets for such malware [23]. For example, most phone operating systems include an API allowing all applications to directly access the address book.
- *Botnet*: A significant portion of current malware activity results in a PC's membership into a botnet. Ikee.B [21] (mentioned above) was the first of what is expected many to come. So called *mobots* (mobile bots) [30] will most likely be similar to those of existing botnets (e.g., providing means of DoS and spam distribution); however, the targets will change. Core telephony equipment is expected to be subject to DoS by phones, and mobot-originated SMS spam will remove the economic disincentive for spammers, making SMS spam much more frequent and wide spread. Finally, the phone functionality will be used. For example, telemarketers could use automated dialers from mobots to distribute advertisements, creating "voice-spam" [31].

2.1.2 Privacy

Not all smartphone threats can be easily classified as malware. As discussed by this and other work [32], legitimate and otherwise benign applications have been found to share privacy sensitive information without acquiring user consent. In Chapters 5 and 6 we discuss how the phone's geographic location is often included in requests to advertisement servers. We also found use of phone identifiers, including the phone number, in network transactions with no apparent need for the information. Unfortunately, not everyone perceives this disclosure as harmful to the user, and developers include this functionality without fully understanding the repercussion to users.[1] Instances are frequently classified as cases of over zealous data gathering [34], as malware implies malicious intent by the developer. Therefore, throughout this document, we use the more generic classification of "dangerous functionality".

2.2 Application Markets

An application market, also known as an application store, or "app store," is a central point for sales, distribution, and discovery of applications. Almost all popular smartphone operating systems have at least one application market, if not more. The largest application markets, by an order of magnitude, are Apple's App Store, and Google's Android Market. These markets provide applications for not only smartphones, but also tablets and personal media players. Recently, Apple announced a variant of its App Store for desktops running Mac OS X [35].

The appeal of application markets is how they simplify sales, distribution, and discovery. The market is responsible for accepting payment from consumers, compensating the application provider, and hosting the applications in a manner that allows discovery by consumers. By eliminating the need to set up payment and distribution infrastructure, the application provider is often a single or small group of largely unknown developers. Furthermore, consumers are more likely to provide the application market with credit card information than an unknown provider.

The application market client runs on the smartphone or other mobile device. It provides a central location for the consumer to find new applications (e.g., via keyword searches). Clients typically display the name of the application, a developer/provider name, a description of the application, and reviews/ratings by other consumers. On some platforms, such as the Apple iPhone iOS, there is only one market.[2] However, on other platforms, e.g., Android, there are multiple markets. In the case of Android, Google's Android Market is considered the "official" market, but others are available for install by consumers.

It is a common misconception that an application market can act as a security filter for all smartphone threats [2]. For example, news articles frequently criticize Google's Android Market, because it does not perform any checking, as is done by Apple's App Store. However, it is unclear what security checks are performed by Apple, if any [36].

---

[1] The EFF discusses the repercussions of location information [33].

[2] Note, "jailbroken" iOS devices also access the Cyndia market.

Performing security checks at the market level are impractical for several reasons. First and foremost, they do not have proper context. Each user has different expectations for what applications should and should not do. The differences are particularly prominent for privacy concerns. Second, markets receive a high volume of new and upgraded applications. Performing more than simple analysis is impractical, if not impossible.

There are, however, clear security advantages of application markets. For example, some markets implement so called "kill switches" that allow the market to not only stunt distribution of dangerous applications, but also remove them from deployed devices [3]. This is an invaluable feature that has not been available for desktop environments. However, the ability to remotely remove applications is a concern in and of itself. Amazon's removal of the book "1984" from Kindle devices is a relevant analogy of the dangers [37]. Nonetheless, kill switches are arguably a net gain for end user security.

Finally, markets allow the consumer to identify the source of an application. For example, the market can ensure that only Company X can have the provider/developer name "Company X". This mitigates threats wherein the adversary distributes an application masquerading as Company X's official application (e.g., to prevent phishing for banks). It can also mitigates threats where the adversary grafts malware onto existing trusted applications. However, the efficacy of the developer name should be viewed with caution. First, just as in PKIs [38], it is difficult to determine who the real Company X is. This is even more difficult for individual developers. Second, multiple application markets remove guarantees that a known developer/provider is the same as the one listed. For example, an Android Trojan was recently distributed in a Chinese application market by grafting the malicious code onto existing applications [39]. Therefore, in the face of multiple markets, the security benefit of developer names is difficult to qualify for end consumers.

2.3 Case Study: Android

The following chapters use the Android smartphone platform to understand security challenges in the domain. The Android platform and its applications are representative of the current state of smartphone technology from which trends can be established. Furthermore, Android is open source, which allows deep understanding of platform operation and enables augmentations to enhance our ability to study applications.

Android is best described as a middleware running on top of embedded Linux. The

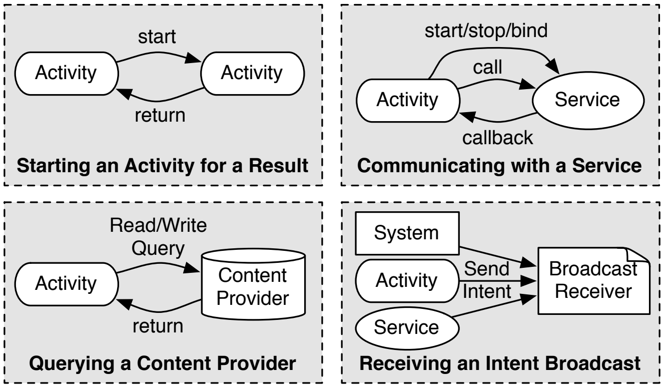

Figure 2.1. Typical IPC between application components

underlying Linux internals have been customized to provide strong isolation. Each application is written in Java and runs as a process with a unique UNIX user identity. This design choice minimizes the impact of a buffer overflows. For example, a vulnerability in web browser libraries [40] allowed an exploit to take control of the web browser, but the system and all other applications remained unaffected.

All interprocess communication (IPC) between applications occurs through *binder*. The binder IPC framework provides the base functionality for both application operation and security enforcement. In the remainder of this section, we describe the application and security frameworks. See Enck et al. [41] for additional explanation and helpful examples.

2.3.1 Application Framework

Android applications are defined as collections of *components*. If necessary, components are automatically started by the system in response to IPC. IPC between components takes the form of *intent* messages (in most cases). Intent messages are addressed to an *action string* used by the application framework to determine the correct recipient. Application and component resolution is performed using *intent filters* defined the developer of the target application. Android defines many standard action strings; however, as long as the source and target applications agree upon an action string, any text string can be used. Alternatively, an intent message can be addressed to a specific component by specifying the target application's namespace, which bypasses the resolution logic. Figure 2.1 depicts typical IPC between components that potentially crosses applications.

- An *activity* component interfaces with the physical user via the touchscreen and keypad. Applications commonly contain many activities, one for each "screen" presented to the user. The interface progression is a sequence of one activity "starting" another, possibly expecting a return value. Only one activity on the phone has input and processing focus at a time.

- A *service* component provides background processing that continues even after its application loses focus. Services also define arbitrary interfaces for remote procedure call (RPC), including method execution and callbacks, which can only be called after the service has been "bound".
- A *broadcast receiver* component acts as an asynchronous event mailbox for intent messages "broadcasted" to an action string. Android defines many standard action strings corresponding to system events (e.g., the system has booted). Developers often define their own action strings.
- A *content provider* component is a database-like mechanism for sharing data with other applications (e.g., an address book). The interface does not use intent messages, but rather is addressed via a *content URI*. The interface supports standard SQL-like queries, e.g., `SELECT`, `UPDATE`, `INSERT`, through which applications access information. The content provider interface also includes IO streams for reading and writing file content.

Every application package includes a *manifest file*. The manifest file specifies all components in an application, including their types and intent filters. All components must be specified in the manifest file, with the exception of broadcast receivers, which can be dynamically created. Note that dynamic broadcast receivers are commonly used by activity components to receive information only while the application is running, and they cannot be started automatically by the system.

2.3.2 Security Framework

An Android phone's security policy is primarily defined in the manifest files of all installed applications, including the applications that comprise the application framework (known as *system applications*). Security policy in the manifest file primarily consists of 1) permissions defined by the application, 2) permissions requested by the application, and 3) the permissions that restrict access to components. Requested permissions are granted to an application at install time and cannot change without reinstalling (e.g., upgrading).

Application developer specified security policy is a result of the nature of mobile phone development. Managing access control policies of hundreds (thousands) of potentially unknown applications is infeasible in many regards. Hence, the Android developers simplified access control policy specification by having developers define permission labels to access their interfaces. The developer does not need to know about all existing (and future) applications. Instead, the permission label allows the developer to indirectly influence security decisions.

Android allows or disallows an application's request for a permission based on the permission's *protection-level*. A permission's protection-level is set by the developer of the application defining the permission. There are four protection-levels. Permissions with *normal* protection-level are always granted to any application that asks for it. Permissions with *dangerous* protection-level are granted upon the user's agreement. The permissions protected with *signature* protection-level are granted only to the application signed with the same certificate as the application that defines the permission. Lastly, the *signatureOrSystem* protection-level is granted using the same logic as the signature protection-level, with the addition that the permission is granted to any application that is *a*) installed in the system firmware image (`/system/app`), or *b*) signed by the key that signed the system firmware. Note that this crude policy is the only vehicle for an application to protect its interfaces (aside from prohibiting any use by labeling them "private", as mentioned below).

Android mediates IPC based on permissions using a middleware reference monitor [42]. Simply put, an application may initiate IPC with a component in another (or the same) application if it has been assigned the same permission label associated with the target component IPC interface. Permission labels are also used to restrict access to certain library APIs. For instance, there is a permission label that is required for an application to access the Internet. Android defines many system permissions in addition to those defined by the applications.

There are a number of extensions to the Android permission system that extend policy expressibility and reduce the complexity of specifying and managing application security.

- *Some policy is defined within application source code, and not the manifest file.* For example, the API for broadcasting intent messages optionally allows the developer to specify a permission label to restrict which applications may receive it. This provides an access control check in the reverse direction of the IPC. A second example is the *checkPermission()* reference monitor hook that can be placed anywhere in an application. These hooks are primarily used to differentiation access to RPC interfaces in service components.
- *The Android middleware does not enforce all permission checks.* Several permission

labels defined by the Android framework (e.g., Internet and Bluetooth permissions) are enforced in the kernel based on UNIX group identifiers. For these special permissions, an application is added to the corresponding UNIX group at install time.

- *The developer is not forced to specify a permission label in the manifest file to restrict access to a component.* If no label is specified, there is no restriction (i.e., default allow).
- *Some components are "private" based on manifest file specification.* If a component is private, it can only be accessed by components defined in the same application. If a component does not define an intent filter, Android automatically makes it private. Alternatively, the developer can explicitly make a component private. Private components simplify security policy specification for developers.
- *Content provider component access control policy can restrict read and write commands separately.* Frequently, an application developer may wish to specify read-only access to data. This is accomplished by defining separate read and write permission labels.
- Pending intent *objects allow another application to complete an IPC.* Applications create pending intent objects by first creating an intent message as if it were performing the IPC, and then acquiring a pending intent object reference specific to the target component type (e.g., activity). In a separate IPC, the pending intent is sent to an application. That application can fill in any address or data fields not specified by the original application. When the application completes the IPC, it operates within the protection domain of the original application. If used correctly, pending intents can enhance an application's security. For example, an application can use a pending intent to allow a system application to start a private activity. However, pending intents provide a limited form of delegation, and can lead to vulnerabilities if used unsafely (e.g., passed to an untrusted application without specifying the intent address completely).
- *Applications can delegate subparts (e.g., individual records) of a content provider.* Using the *URI permission* feature, any application with read or write access to a content provider can delegate that access for a specific record (or URI path) to an application that does not have permission. This feature can enhance application and system security by allowing least privilege protection. However, this introduction of runtime delegation complicates both policy management and analysis.

Note that URI permissions can only be delegated if the content provider definition specifically allows the feature.

- *Special system defined broadcast actions cannot be spoofed.* Following the discovery of an application's ability to forge the receipt of an SMS message by broadcasting an intent message to a specific action string (a result from work [5] contributing to this dissertation), the Android system developers added a new permission label allowing the recipient of the SMS intent broadcast to ensure it was sent by the system. However, there are many intents broadcast by the system, and preventing forging attacks using permissions requires significant policy management by application developers. Therefore, the Android reference monitor includes the concept of "protected broadcasts." That is, within the framework, a set of action strings are specified as "protected" to ensure only the system (and not third-party applications) can broadcast intents with those actions.

Chapter 3

Related Work

This dissertation studies security in mobile operating systems with a specific focus on smartphones. As previously discussed, these operating systems are the result of the Internet pushing computing technology towards a pervasive, service-oriented architecture. While much of the following chapters focus on the study of applications, the purpose of these studies is to reflect upon the impact of observed phenomena on the realities of operating system level protection for this emerging platform. We begin the following discussion by highlighting the foundations of operating systems security.

3.1 Operating System Security

Operating systems provide the foundations for security in computing systems. OS security is a very broad and deep area of computer science. Many of the underlying concepts of OS security were developed during the design of Multics [43] during the 1960's and '70s. In 1974, Salter [44] enumerated nine areas of active OS security research, including: system penetration, user interface studies, mathematical models, and protection mechanisms. A brief survey of current OS security literature demonstrates a continued presence of these themes and hence the difficulty of OS security problems. It is, to say the least, a very intricate and challenging subject. Jaeger [45] provides an overview of OS security fundamentals and how they relate to both research and production systems spanning from Multics to today's UNIX and Microsoft Windows platforms.

In traditional operating systems, applications execute as *processes*. Each process has a *protection domain*, defined as the information and resources that the process may access. Protection in operating systems can be represented as an *access control matrix* [46].

The matrix is made up of a set of *subjects* (e.g., processes) and *objects* (e.g., files) with matrix elements populated by *actions* (e.g, read, write) that subjects may perform on objects. Because access control matrices are often sparsely populated, protection policy commonly takes the form of an *access control list* (ACL). For this representation, each object is given an ACL defining the subjects and the actions those subjects may perform. Another common representation of protection policy is a *capability list*, or C-List. Here, subjects are assigned a C-List defining the objects and the actions the subject may perform on them.

Access control polices can be either *discretionary* or *mandatory*. Discretionary policy is managed by the subjects. Traditionally, subjects are users, or rather, processes executing with the authority of a user. Therefore, discretionary access control policy is commonly described as a policy managed by the user. If policy transitions are not carefully limited, security goals can be circumvented. The decidability of whether or not discretionary policy can reach an unsafe state is unknown, This condition is known as the *safety problem* [47]. Mandatory access control policy, or MAC policy, restricts policy management to an administrative entity, and is therefore not subject to the safety problem. Much of operating systems security literature has focused on MAC, as it can provide provable security guarantees (e.g., with respect to information flows).

At the foundation of MAC policy enforcement is a *reference monitor* [42]. From a purist perspective, a reference monitor requires: 1) *complete mediation* of all security sensitive operations; 2) *tamperproofness* of the enforcement mechanism; and 3) *verifiability* of the complete mediation and tamperproofness. However, few systems actually achieve or attempt rigorous verifiability due to practical limitations. Hence, programming errors that lead to privilege escalation or circumvention often go unnoticed until it is too late.

3.1.1 Kernel-level Protection

Verifiability was a primary focus of OS security research in the late 1970's and '80s. To be verifiable, the trusted computing base of the system must be small. This observation led rise to the concept of a *security kernel*, which can be defined as the hardware and software necessary to achieve a reference monitor. Early security kernels include Scomp [48] and GEMSOS [49]. Scomp leverages custom hardware features to minimize trusted software. For example, protection rings to isolate trusted and untrusted software (a Multics concept) are managed entirely in hardware, and a security protection module mediates direct memory access (DMA) on the I/O bus. In contrast, GEMSOS was designed to operate on commodity x86 hardware. While the x86 architecture provides protection rings, it cannot mediate DMA. Despite these limitations, GEMSOS was formally validated and verified to the Trusted Computer System Evaluation Criteria's level A1 [49].

As research of verifiable security kernels progressed, Rushby [50] observed that security goals of the verification (in this case, multilevel security) were often at odds with the practical realities, resulting in many trusted processes outside of the kernel. This observation lead to the concept of a *separation kernel* [51]. While a separation kernel is, in fact, a security kernel, it has the very specific security goal of achieving separation tantamount to hosts in a distributed system. Hence, verification of the separation kernel simplifies to ensuring isolation between "regimes." In this model, regimes need not be completely isolated, but communication channels must be well defined, and with the ability to "cut" them if needed.

The concept of a separation kernel led way to microkernels, which remove all unnecessary functionality from the kernel, placing it in user-space processes. The Mach microkernel [52] applies this approach to the UNIX architecture, providing privilege separation of kernel functionality. However, the design increases IPC latency, and the associated context switches and accounting causes poor performance. The L4 microkernel [53, 54] showed that microkernel performance need not be significantly worse than macrokernel architectures using processor-specific implementations of processor-independent abstractions.

In his description of the separation kernel model, Rushby [50] noted the similarity of separation kernels to virtual machine monitors (VMMs) [55, 56], e.g., the IBM VM/370 [57]. In this environment, the VMM simulates "bare hardware" to allow multiple virtual machines to simultaneously run on the same physical machine. Kelem and Reiertag [58] formally model the separation model for VMMs. More recently, commodity VMM systems such as VMware [59] and Xen [60] have been used provide a security perimeter. Terra [61] demonstrates an architecture where desktop applications are run in separate VMs. sHype [62] implements mandatory access control for communication between VMs.

3.1.2 Information Flow

Historically, mandatory access control has been synonymous with information flow control (IFC). IFC is different from traditional access control, because it reasons about the transitive access of information. Similar to Lampson's access control matrix [46], IFC models a set of subjects S and a set of objects O. However, read and write actions are represented as flows ($\rightarrow$). For example, when a subject $s \in S$ reads from an object $o \in O$, then $s \leftarrow o$. Conversely, when s writes to o, then $s \rightarrow o$. These operations form an information flow graph $G = (V, E)$ where the vertices V are the union of subjects and objects ($V = S \cup O$) and the edges E are flows. Note that Lampson's access matrix can be converted into an information flow graph by mapping each action type to one of read, write, or both read and write.

Information flow security models label each subject and object with a security class. Denning [63] organized security classes into a lattice to define verifiable security goals. The lattice specifies allowable flows between security classes. Bell and LaPadula [64] define the multi-level security (MLS) model using lattices for confidentiality. In MLS, the lattice encodes the simple-security (no read up) and $\star$-security (no write down) policies over hierarchical security classes (e.g., top-secret, secret, confidential, and unclassified). Biba [65] proposes the dual for integrity, encoding the simple-integrity (no read down) and $\star$-integrity (no write up) policies within the lattice. While confidentiality and integrity information flow models can be enforced simultaneously, disjoint sets of security labels are used, otherwise, subjects can only read and write within their own security classes. For example, whereas MLS uses military security classes, an integrity model might use security classes such as: trusted, system, application, user, and untrusted.

While the MLS and Biba models provide mathematically provable guarantees, practical implementations rely on the security of trusted processes. A trusted process is allowed to operate outside the confines of the security lattice and therefore represents an attack surface. The Clark-Wilson [66] integrity model constrains how trusted processes can exist. It provides verifiability through certification of all trusted processes. The CW-lite [67] model relaxes Clark-Wilson by not requiring certification of entire processes, but rather relies on defined filtering interfaces to upgrade or discard inputs read from low integrity processes. Usable Mandatory Integrity Protection (UMIP) [68] also relaxes stricter integrity models and defines trust in terms of types of information flows (e.g., IPC, network). UMIP handles exceptions to a low watermark model using rules informed by the system's discretionary access control configuration. A third practical integrity model, Practical Proactive Integrity (PPI) [69], uses a combination of integrity labels and policies to flexibly guarantee system integrity.

Decentralized information flow control (DIFC) [70, 71, 72] is another approach to overcoming the limitations of trusted processes. DIFC is an application of the decentralized label model (DLM) [73] for operating systems. Instead of defining label exceptions to a centrally defined MLS or Biba lattice, DIFC breaks trust into many non-comparable security classes. These classes, called *tags*, are created and managed by applications. Subject and object labels consist of sets of tags, and the enforced security lattice is defined by a partial order of the set of tags. A subject is said to "own" a tag if it can add and remove it from its label (effectively declassifying or endorsing information). Furthermore, subjects can delegate add and remove tag capabilities to other subjects to manage trust. Fundamental to this approach is the observation that a system has many types of non-comparable information, and frequently, only the application developer has sufficient context to administer protection policy.

3.1.3 Other MAC Models

Not all MAC protection systems focus on information flow. For example, Domain Type Enforcement (DTE) [74] provides privilege separation for root owned processes on a UNIX system. DTE is similar to Type Enforcement (TE) [75]; however, in DTE, only objects are labeled with types. Subjects are assigned a domain, which is a tuple containing 1) "entry point" programs (i.e., a binary path), 2) access rights (i.e., read, write, execute) to object types, and 3) access rights (i.e., signals) to other domains. SELinux [76] (an implementation of Flask [77] for Linux) is also based on types. Like DTE, SELinux only uses types for objects; however, subjects are assigned a role. SELinux roles are similar to, but distinct from, roles in RBAC [78]. SELinux policy specifies how roles can act on types, as well as transition rules between roles. Strict SELinux policy defining access control for all processes is very complex and often breaks systems, therefore, a target policy mode is provided. Similar to DTE, the SELinux targeted policy mode provides root process privilege separation. In response to the complexity of SELinux policy, AppArmor [79] (formerly known as SubDomain [80]) defines MAC policy using path names to achieve semantics similar to SELinux's targeted mode.

Capabilities have also been used for MAC protection. A capability is similar to a physical key that is given to a subject to gain access to some interface. In a capability model, protection policy is managed though the delegation of capabilities from one process to another. Capabilities naturally encode the *principle of least privilege* [81], as access policy can follow the runtime context. This characteristic is most well known as a solution to the *confused deputy problem* [82]. This problem occurs when a untrusted process convinces trusted system process to performs an action on its behalf. In a capability model, the system process need only perform actions using the capability set assigned to untrusted processes.

While capability delegation appears discretionary in nature, it has been used to enforce MAC in operating systems. Most notable early OS based upon capabilities is HYDRA [83]. Boebert [84] and Karger [85] note that traditional capability systems fail to ensure the $\star$-security property (no write down) required by many MAC systems. The failure arises from the ability for a high secrecy process to read the capabilities given to a low secrecy process and then use those capabilities to write to a low secrecy interface. The SCAP [86] and EROS [87] operating systems overcome this property using *read-only* capabilities and *weak* capabilities, respectively.

3.1.4 Defense of User Information

Trojans and similar malware typically enter a system as downloaded files. Many techniques based on sandboxing have been proposed to prevent malware from harming the user. Janus [88] defines secure profiles for helper applications (e.g., document viewers and media players) to limit the effects of exploitation by malicious files downloaded by Email clients and Web browsers. Jaeger et al. [89] flexibly limit access provided downloaded executable content (e.g., scripts) based on policy specific to its source. MAPbox [90] locates predefined sandbox policies based on application labels. Lai and Gray [91] confine applications based on program arguments and an assigned trust level. Exceptional file accesses are presented to the user for authorization.

Models where the user specifically designates an action as untrusted have also been proposed. TRON [92] uses capabilities for files and directory trees. Capabilities can be dropped on process fork to limit the protection domain of untrusted applications. Plash [93] provides similar functionality to users within a command line shell interface.

Damage by malware and vulnerable applications can also be mitigated by running applications as different identities. For example, Polaris [94] uses predefined profiles that automatically execute specified applications as a separate user account. The concept of program level identities for file access originally appeared in PACLs [95], where program access control lists are specified on files. Several systems treating programs as first-class principals have since been proposed, including PinUP [96, 97], Sub-Operating Systems [98], Sub-Identities [99], and FileMonster [100]. Note that by running all applications as separate users, Android's file access control also falls within this category.

3.2 Smartphone Security

3.2.1 Smartphone OS Protection

Smartphones are often considered as resource constrained devices running full featured operating systems. As such, researchers have reapplied traditional techniques with the modifications to meet the requirements of the new environment. Integrity measurement and remote attestation [101] is one such area. Smartphones have multiple stakeholders. The Trusted Computing Group's (TCG) mobile specification [102] assumes four stakeholders: device manufacturer, network operator, third-party service provider, and end user. It uses a Mobile Remote-Owner Trusted Module (MRTM) and a Mobile Local-Owner Trusted Module (MLTM) to perform duties traditionally performed by a Trusted Platform Module (TPM). A TCG compliant phone architecture has multiple isolated domains, one for each stakeholder, where each domain has its own logical MRTM, with the exception of the end-user domain, which uses an MLTM. Zhang et al. [103] realize the TCG's trusted mobile phone specification using SELinux to isolate operational domains. More recently, Zhang et al. [104] design an efficient remote attestation framework for the LiMo platform. Similarly, Nauman et al. [105] implement remote attestation for Android, including measurement of applications. Integrity measurement has also been used to achieve more classical goals. For example, Muthukumaran et al. [106] enhanced the Linux-based Openmoko platform with integrity measurement and SELinux to enforce CW-lite security goals in the software installer. Shabtai et al. [107] ported SELinux to Android to harden the Linux-based system beneath the middleware.

Virtualization has been applied to phones to provide coarse separation between sets of applications. The VMware mobile virtualization [108] allows an enterprise administrator to install a hosted VM containing business applications. However, this threat model allows vulnerabilities in the user's personal phone installation to compromise the business VM. In contrast, the OK Labs [109] uses L4 as a hypervisor to simultaneously run Android, Symbian, and Windows. Finally, Lee et al. [110] enforce MAC policies for communication between VMs running on a phone. The architecture has similarities to a configuration of ARM TrustZone for embedded Linux [111].

Higher-layer application security policy frameworks have also been considered. Mulliner et al. [112] prevent attacks by labeling processes based on the security sensitive interfaces they access. Policies prevent cross-service attacks by preventing a process from using unsafe combinations of interfaces. Ion et al. [113] extend the J2ME security framework to include policies to limit service use during a specified time period (e.g., SMS, data).

SxC [114] allows users (or experts) to define “security contracts” for .NET applications running on a Windows Mobile phone. This approach allows users to install untrusted applications, but retain certain runtime guarantees. In a related effort on the Android platform, Apex [115] changes the “all or nothing” permission model, allowing users to select individual permissions granted to applications as well as conditions (e.g., time of day) when permissions may be used. CRePE [116] also enforces fine-grained user defined contextual polices (e.g., location, time). In contrast, Saint [117] enhances the expressibility of developer-defined security policies for the Android platform. Saint policies can restrict both application installation and runtime interactions between applications based on configuration (e.g., permissions held by other applications), signatures (e.g., whitelisting and blacklisting), and environmental context (e.g., location). Porscha [118] enforces policy for content owners, restricting how SMS and Email messages can be used once arriving on the phone.

Karlson et al. [119] interviewed smartphone users to better understand their willingness to share handsets with other users. Their findings indicate that smartphones should provide less-privileged modes to allow “guest users” to use phones without compromising the owners privacy. Similar findings are reported by Liu et al. [120], and their xShare prototype for Windows Mobile demonstrates how such an environment should operate. DiffUser [121] provide similar support for Android.

Similar to our work, researchers have studied the limitations of application-level policy frameworks. Shin et al. [122] formally model Android’s security policy language an interactions between applications. Using this model, they identify a privilege escalation attack where an unprivileged application can perform security sensitive operations either by exploiting vulnerabilities in other applications or collusion [123]. One solution to this problem requires transitive tracking of permission use. Chaudhuri [124] defines a language based approach to evaluating communication between applications. This formal model is used to build the ScanDroid [125] framework for automatically certifying applications based on their source code. However, this model requires a partial ordering based on existing Android permissions. Unfortunately, most Android permissions are non-comparable, which severely impedes the practical application of this approach.

3.2.2 Malware Detection

The detection and prevention of Trojans and other malware has long been a topic of security research. In desktop and server environments, antivirus software commonly scans files to detect malware based on predefined signatures. However, such scanning is too resource intensive for smartphones, specifically with respect to energy consumption. Therefore, alternative strategies are required.

Venugopal and Hu [126] describe an efficient signature base scanner optimized for mobile phones. As an alternative, Bose et al. [127] propose the use of behavioral analysis based on security-sensitive operations. Andromaly [128] implements anomaly detection for Android based on runtime artifacts such as CPU load, input events, and energy consumption. Nash et al. [129] consider an intrusion detection system for battery exhaustion attacks; however, the work only discusses potential algorithms. Finally, Kim et al. [130] use an energy consumption based anomaly detection method to detect malware. The proposed model supports a stand alone model where all analysis occurs on the phone, as well as the use of a remote server for processing data.

Several proposed systems send logs of system events to a central server for analysis. However, as noted by Miettinen et al. [131], the logging mechanism must be careful to protect the user's privacy, ensuring that privacy sensitive information never leaves the device. SmartSiren [27] sends logs of device activities such as use of Bluetooth and SMS to a central server. The central server then performs behavioral analysis across devices, which is advantageous for detecting Bluetooth worms. Similarly, Schmidt et al. [132] record the amount of free RAM, user activity, process count, CPU usage, and the number of sent SMS messages for analysis by a central server.

Malware analysis can also be performed by maintaining consistency between a smartphone and a replicated image on a network server [133]. The Paranoid Android [134] takes this approach, but conserves energy by using "loose synchronization" to only send data when the user is using the device. Oberheide et al. [135] provide an alternative model for server side antivirus scanning. Here, the authors create a mobile client for the CloudAV [136] architecture that uploads files to a server for scanning if a file with the same cryptographic hash has not been scanned.

3.3 Information Tracking

The operating system level information flow control described in Section 3.1.2 tracks information at a coarse granularity. DEFCon [137] uses a logic similar to the DIFC-based operating systems, but focuses on events and modifies a Java runtime with lightweight isolation. Related to these approaches, PRECIP [138] labels both processes and shared kernel objects such as the clipboard and display buffer. However, these process-level information flow models are coarse grained and cannot track sensitive information *within* untrusted applications.

Language-based information flow security [139] extends existing programming languages by labeling variables with security attributes. Compilers use the security labels to generate security proofs, e.g., Jif [140, 73] and SLam [141]. Laminar [142] provides DIFC guarantees based on programmer defined security regions. However, these languages require careful development and are often incompatible with legacy software designs [143].

Dynamic taint analysis (also known as "taint tracking") provides information tracking for legacy programs. In taint tracking, variables or registers are *marked* at a taint source where the security semantics of the information are known. These taint markings are propagated dynamically during execution and eventually observed at a taint sink, where appropriate actions are invoked based on the marking. Taint tracking propagation traditionally uses definition data flow semantics, also know as explicit flows. This technique relies on an inductive definition for each instruction that effects the flow of information. For example, the instruction `c = a + b` assigns the markings on `a` and `b` to `c`. However, data flow semantics does not capture all information flows. Implicit flows, also known as control flows, result from branch and loop instructions. For example, a variable `y` can be set to `1` if a monitored variable `x` matches a specific value. Here, there is a flow of information from `x` to `y` without an explicit instruction. Dynamic taint tracking systems account for some types of explicit flows. For example, the marking on the index in an array access is often propagated to account for translation tables (e.g., ASCII to UNICODE conversion). Taint scopes are also sometime used when the start and end instruction addresses are known for branches and loops. Here, the marking on conditional variables is propagated to all variables assigned inside of the branch or loop scope. However, this is an approximation and can lead to significant false positives. Finally, to capture for all implicit flows, some static analysis is required [144].

Dynamic taint tracking has been used to enhance system integrity (e.g., defend against software attacks [145, 146, 147]) and confidentiality (e.g., discover privacy exposure [148, 149, 150]), as well as track Internet worms [151]. When enhancing system integrity, the network interface is used as a taint source, marking all inbound information as untrusted. If the untrusted information is not explicitly sanitized before use in a control point (e.g., the stack return address). When used to monitor privacy, the network interface is the taint sink. Here, outbound traffic is inspected for taint markings assigned at privacy sensitive sources. Note that privacy sensitive sources are not always well defined. For example, it is difficult to programmatically distinguish between the characters entered for a password or credit card number from the contents of an email at the keyboard input device interface. However, as we discuss in Chapter 5, smartphones have several well defined privacy taint sources with clear semantics for all information read from the interface.

Dynamic tracking approaches range from whole-system analysis using hardware extensions [152, 153, 154] and emulation environments [155, 148] to per-process tracking using dynamic binary translation (DBT) [156, 146, 147, 150]. The performance and memory overhead associated with dynamic tracking has resulted in an array of optimizations, including optimizing context switches [146], on-demand tracking [157] based on hypervisor introspection, and function summaries for code with known information flow properties [150]. If source code is available, significant performance improvements can be achieved by automatically instrumenting legacy programs with dynamic tracking functionality [158, 159]. Automatic instrumentation has also been performed on x86 binaries [160], providing a compromise between source code translation and DBT. Our TaintDroid design discussed in Chapter 5 was inspired by these prior works, but addressed different challenges unique to mobile phones. To our knowledge, TaintDroid is the first taint tracking system for a mobile phone and is the first dynamic taint analysis system to achieve practical system-wide analysis through the integration of tracking multiple data object granularities.

Finally, dynamic taint analysis has been applied to virtual machines and interpreters. Haldar et al. [161] instrument the Java String class with taint tracking to prevent SQL injection attacks. WASP [162] has similar motivations; however, it uses positive tainting of individual characters to ensure the SQL query contains only high-integrity substrings. Chandra and Franz [163] propose fine-grained information flow tracking within the JVM and instrument Java byte-code to aid control flow analysis. Similarly, Nair et al. [164] instrument the Kaffe JVM. Vogt et al. [165] instrument a Javascript interpreter to prevent cross-site scripting attacks. Xu et al. [158] automatically instrument the PHP interpreter source code with dynamic information tracking to prevent SQL injection attacks. Finally, the Resin [166] environment for PHP and Python uses data flow tracking to prevent an assortment of Web application attacks. When data leaves the interpreted environment, Resin implements filters for files and SQL databases to serialize and de-serialize objects and policy with byte-level granularity. TaintDroid's interpreted code taint propagation bears similarity to some of these works. However, TaintDroid implements system-wide information flow tracking, seamlessly connecting interpreter taint tracking with a range of operating system sharing mechanisms.

3.4 Security and Privacy Analysis

Many tools and techniques have been designed to identify security concerns in software. The following discussion overviews application of these techniques on “real code” as means of enhancing system security (as opposed to system penetration). We begin with studies targeting vulnerability analysis.

3.4.1 Vulnerability Analysis

Software written in C is particularly susceptible to programming errors that result in vulnerabilities. Ashcraft and Engler [167] use compiler extensions to identify errors in range checks. The compiler modifications are used to identify over 100 vulnerabilities in the Linux and OpenBSD kernels. More general vulnerabilities have been identified using MOPS [168], a model checking program analysis tool that defines vulnerabilities as finite state automata (FSA). Chen et al. [168] used MOPS to analyze eight popular Linux applications, consisting of over a million lines of code. In several applications, they identify failures to completely drop root privileges. Schwarz et al. [169] also use MOP, checking all applications in a Linux distribution (60 million lines of code), and discovering 108 exploitable bugs. In related work, Ball and Rajamani [170] use SLAM [171] to discover errors in Windows XP device drivers.

Java applications are inherently safer than C applications and avoid simple vulnerabilities such as buffer overflows. Ware and Fox [172] compare eight different open source and commercially available Java source code analysis tools. They find that no one tool detects all vulnerabilities. Hovemeyer and Pugh [173] study six popular Java applications and libraries using FindBugs extended with additional checks. While analysis included non-security bugs, the results motivate a strong need for automated analysis by all developers. Livshits and Lam [174] focus on Java-based Web applications. In the Web server environment, inputs are easily controlled by an adversary, and left unchecked can lead to SQL injection, cross-site scripting, HTTP response splitting, path traversal, and command injection. Using static taint analysis, nine server side Java applications are studied, finding 41 potential security violations, of which 29 were security errors. Felmetsger et al. [175] also study Java-based web applications; however, they advance vulnerability analysis by providing automatic detection of application-specific logic errors. Their tool, Wailer, uses existing dynamic analysis tools to identify likely invariants. False invariants are then removed using model checking. No manual annotation is required in either stage. Wailer is validated by studying four popular Web applications, identifying 30 vulnerabilities.

Web applications are also frequently written in PHP. Jovanovic et al. [176] design Pixy to identify SQL injection and cross-site scripting vulnerabilities using static taint analysis for PHP. They use Pixy to study seven popular PHP applications and observe a low false positive rate, identifying over 200 vulnerabilities. Balzarotti et al. [177] design Saner to discover input validation vulnerabilities in PHP applications. Saner uses both static and dynamic analysis. First, static analysis conservatively models modifications to inputs on all paths from the source to a sink. Next, dynamic analysis removes false positives by determining which code paths are used by the application. They use Saner to study five popular PHP applications and identify 13 new vulnerabilities.

3.4.2 Privacy and Malicious Behavior Analysis

The spyware class of malware seeks to extract private information. Browser Helper Objects (BHOs) and Web browser toolbars are common sources of spyware. Kirda et al. [178] consider behavioral properties of BHOs and toolbars. They observe that to achieve its goal, BHO and toolbar spyware must both monitor user behavior and invoke Windows API calls that potentially leak information. Using static analysis of API calls, with dynamic analysis to refine executed code, they study 33 malicious and 18 benign BHOs and toolbars. Using the combined analysis, only two studied components are incorrectly identified as spyware. Egele et al. [149] target information leaks by browser-based spyware explicitly using dynamic taint analysis. Their approach modifies QEMU to only perform taint tracking of BHOs. The study investigates 21 known spyware components and 14 benign BHOs. In addition to the spyware components, the study found two benign BHOs actually leak sensitive data.

Yin et al. [148] consider privacy-breaching malware in general with Panorama, a tool designed for whole-system, fine-grained taint tracking. Panorama creates "taint graphs" that are analyzed by malware detection policies. They use Panorama to study 42 real-world malware samples and 56 benign applications. Of these samples, only three were falsely identified as malware due to behavior that was representative information misuse. The authors note that these false positives result from a limitation of using information tracking: namely, the inability to capture intent.

Jung et al. [179] use differential black box fuzz testing in the design of Privacy Oracle. This technique combines controlled input parameters (e.g., a set of usernames) with traditional fuzz testing. Correlations between the controlled inputs and consistent network traffic data is are used as indicators of privacy leaks. The authors use Privacy

Oracle to study the top 20 applications from `download.com` and the 6 most popular IM clients, identifying many previously undisclosed leaks. Yumerefendi et al. [180] propose a similar approach for TightLip; however, no application study is performed.

Finally, Egele et al. [32] use PiOS to perform static analysis on iOS applications for the iPhone. PiOS reconstructs information flows in Objective-C binaries. The authors use PiOS to study the use of device ID, location, address book, phone number, browser history, and photos in over 1,400 iPhone applications from both the official Apple App Store and the Cydia third-party application market for jailbroken devices. The study found that the majority of the applications leak the device ID. The study also found over half of the applications include advertisement and analytics libraries, which caused difficulties when deciding of the information flows were a result of the library or the application. To address this, known flows in advertisement and analytic libraries were whitelisted.

Chapter 4

Configuration-level Analysis of Smartphone Applications

Traditional OS protection systems rely on security policy. By analyzing the security policy governing applications, an administrator can determine what an application can do at runtime. In smartphones, security policy is frequently based on permissions presented to the user. As discussed in Section 2.3, the security policy defining what an Android application can do is specified in a manifest file that accompanies the application package. In this chapter, we evaluate application security by asking the question, *what permissions do applications request*? From this, we develop practical security invariants to prevent certain types of malware and dangerous behavior.

4.1 Lightweight Smartphone Application Certification

Application markets have made users comfortable downloading and running smartphone applications. As this continues to increase, so does the potential for user-installed malware. The most effective phone malware mitigation strategy to date has been to ensure only approved software can be installed. Here, a certification authority (e.g., Symbian-Signed) devotes massive resources towards source code inspection. This technique can prevent both malware and general software misuse. For instance, software desired by the end user may be restricted by the service provider (e.g., VoIP and "Bluetooth tethering" applications). However, manual certification is imperfect. Malware authors have already succeeded in socially engineering approval [181]. In such cases, authorities must resort to standard revocation techniques.

We seek to mitigate malware and other software misuse on smartphones without burdensome certification processes for each application. Instead, we perform lightweight certification at time of install using a set of predefined security rules. These rules decide whether or not the security configuration bundled with an application is safe. We focus our efforts on the Android platform, because it: 1) bundles useful security information with applications, 2) is representative of current and future trends for smartphone computing, and 3) is open source, allowing deep investigation and experimentation.

In this chapter, we propose the Kirin[1] security service for Android. Kirin provides practical lightweight certification of applications at install time. Achieving a practical solution requires overcoming multiple challenges. First, certifying applications based on security configuration requires a clear specification of undesirable properties. We turn to the field of security requirements engineering to design a process for identifying Kirin security rules. However, limitations of existing security enforcement in Android makes practical rules difficult to define. Second, we define a security language to encode these rules and formally define its semantics. Third, we design and implement the Kirin security service within the Android framework.

Kirin's practicality hinges on its ability to express security rules that simultaneously prevent malware and allow legitimate software. Adapting techniques from the requirements engineering, we construct detailed security rules to mitigate malware from an analysis of applications, phone stakeholders, and systems interfaces. We evaluate these rules against a subset of popular applications in the Android Market. Of the 311 evaluated applications spanning 16 categories, 10 were found to assert dangerous permissions. Of those 10, 5 were shown to be potentially malicious and therefore should be installed on a personal smartphone with extreme caution. The remaining 5 asserted rights that were dangerous, but were within the scope of reasonable functional needs (based on application descriptions). Note that this analysis flagged about 1.6% applications at install time as potentially dangerous. Thus, we show that even with conservative security policy, less than 1 in 50 applications needed any kind of involvement by phone users.

Kirin provides a practical approach towards mitigating malware and general software misuse in Android. In the design and evaluation of Kirin, this paper makes the following contributions:

- *We provide a methodology for retrofitting security requirements in Android.* As a secondary consequence of following our methodology, we identified multiple vulnerabilities in Android, including flaws affecting core functionality such as SMS

---

[1] Kirin is the Japanese animal-god that protects the just and punishes the wicked.

and voice.

- *We provide a practical method of performing lightweight certification of applications at install time.* This benefits the Android community, as the Android Market currently does not perform rigorous certification.
- *We provide practical rules to mitigate malware.* These rules are constructed purely from security configuration available in application package manifests.

The remainder of this chapter is organized as follows Section 4.2 overviews the Kirin security service and software installer. Section 4.3 presents our rule identification process and sample security rules. Section 4.4 describes the Kirin Security Language and formally defines its semantics. Section 4.5 describes Kirin's implementation. Section 4.6 evaluates Kirin's practicality. Section 4.7 presents discovered vulnerabilities. Section 4.8 concludes.

4.2 Kirin Overview

The overwhelming number of existing malware requires manual installation by the user. While Bluetooth has provided the most effective distribution mechanism [10], as bulk data plans become more popular, so will SMS and email-based social engineering. Recently, Yxe [182] propagated via URLs sent in SMS messages. While application stores help control mass application distribution, it is not a complete solution. Few (if any) existing phone malware exploits code vulnerabilities, but rather relies on user confirmation to gain privileges at installation.

Android's existing security framework restricts permission assignment to an application in two ways: user confirmation and signatures by developer keys. These permissions are referred to as "dangerous" and "signature" permissions, respectively (as discussed in Section 2.3). Android uses "signature" permissions to prevent third-party applications from inflicting harm to the phone's trusted computing base.

The Open Handset Alliance (Android's founding entity) proclaims the mantra, "all applications are created equal." This philosophy promotes innovation and allows manufacturers to customize handsets. However, in production environments, all applications are *not* created equal. Malware is the simplest counterexample. Once a phone is deployed, its trusted computing base should remain fixed and must be protected. "Signature" permissions protect particularly dangerous functionality. However, there is a trade-off when deciding if permission should be "dangerous" or "signature." Initial Android-based production phones such as the T-Mobile G1 are marketed towards

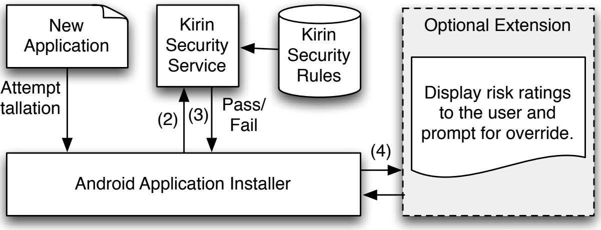

Figure 4.1. Kirin based software installer

both consumers and developers. Without its applications, Android has no clear competitive advantage. Google frequently chose the "feature-conservative" (as opposed to "security-conservative") route and assigned permissions as "dangerous." However, some of these permissions may be considered "too dangerous" for a production environment. For example, one permission allows an application to debug others. Other times it is combinations of permissions that result in undesirable scenarios (discussed further in Section 4.3).

Kirin supplements Android's existing security framework by providing a method to customize security for production environments. In Android, every application has a corresponding security policy. Kirin conservatively certifies an application based on its policy configuration. Certification is based on security rules. The rules represent templates of undesirable security properties. Alone, these properties do not necessarily indicate malicious potential; however, as we describe in Section 4.3, specific combinations allow malfeasance. For example, an application that can start on boot, read geographic location, and access the Internet is potentially a tracker installed as premeditated spyware (a class of malware discussed in Section 2.1). It is often difficult for users to translate between individual properties and real risks. Kirin provides a means of defining dangerous combinations and automating analysis at install time.

Figure 4.1 depicts the Kirin based software installer. The installer first extracts security configuration from the target package manifest. Next, the Kirin security service evaluates the configuration against a collection of security rules. If the configuration fails to pass all rules, the installer has two choices. The more secure choice is to reject the application. Alternatively, Kirin can be enhanced with a user interface to override analysis results. Clearly this option is less secure for users who install applications without understanding warnings. However, we see Kirin's analysis results as valuable input for a rating system similar to PrivacyBird [183] (PrivacyBird is a web browser plug-in that helps the user understand the privacy risk associated with a specific website by interpreting its P3P policy). Such an enhancement for Android's installer provides a distinct advantage over the existing method of user approval. Currently, the user is shown a list of all requested potentially dangerous permissions. A Kirin based rating system allows the user to make a more informed decision. Such a rating system requires careful investigation to ensure usability. This paper focuses specifically on *identifying* potential harmful configurations and leaves the rating system for future work.

4.3 Kirin Security Rules

The malware threats and the Android architecture introduced in the previous sections serve as the background for developing Kirin security rules to detect potentially dangerous application configurations. To ensure the security of a phone, we need a clear definition of a secure phone. Specifically, we seek to define the conditions that an application must satisfy for a phone to be considered safe. To define this concept for Android, we turn to the field of security requirements engineering, which is an off-shoot of requirements engineering and security engineering. The former is a well-known fundamental component of software engineering in which business goals are integrated with the design. The latter focuses on the threats facing a specific system.

Security requirements engineering is based upon three basic concepts. 1) *functional requirements* define how a system is supposed to operate in normal environment. For instance, when a web browser requests a page from a web server, the web server returns the data corresponding to that file. 2) *assets* are "...entities that someone places value upon" [184]. The webpage is an asset in the previous example. 3) *security requirements* are "...constraints on functional requirements to protect the assets from threats" [185]. For example, the webpage sent by the web server must be identical to the webpage received by the client (i.e., integrity).

The security requirements engineering process is generally systematic; however, it requires a certain level of human interaction. Many techniques have been proposed, including SQUARE [186, 187], SREP [188, 189], CLASP [190], misuse cases [191, 192], and security patterns [193, 194, 195]. Related implementations have seen great success in practice, e.g., Microsoft uses the Security Development Lifecycle (SDL) for the development of their software that must withstand attacks [196], and Oracle has developed OSSA for the secure software development of their products [197].

Commonly, security requirements engineering begins by creating functional requirements. This usually involves interviewing stakeholders [186]. Next, the functional re-

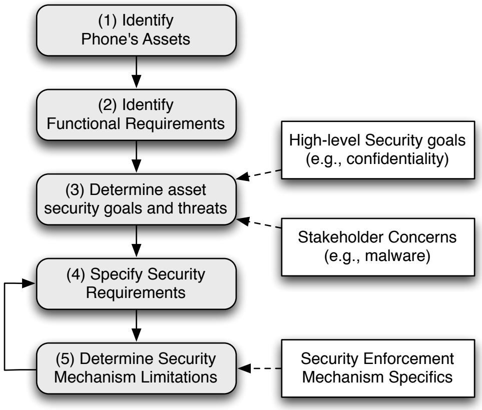
(1) Identify Phone's Assets
(2) Identify Functional Requirements
(3) Determine asset security goals and threats
(4) Specify Security Requirements
(5) Determine Security Mechanism Limitations
High-level Security goals (e.g., confidentiality)
Stakeholder Concerns (e.g., malware)
Security Enforcement Mechanism Specifics

Figure 4.2. Procedure for requirements identification

quirements are translated into a visual representation to describe relationships between elements. Popular representations include use cases [192] and context diagrams using problem frames [198, 185]. Based on these requirements, assets are identified. Finally, each asset is considered with respect to high level security goals (e.g., confidentiality, integrity, and availability). The results are the security requirements.

Unfortunately, we cannot directly utilize these existing techniques because they are designed to supplement system and software development. Conversely, we wish to retrofit security requirements on an existing design. There is no clearly defined usage model or functional requirements specification associated with the Android platform or the applications. Hence, we provide an adapted procedure for identifying security requirements for Android. *The resulting requirements directly serve as Kirin security rules.*

4.3.1 Identifying Security Requirements

We use existing security requirements engineering techniques as a reference for identifying dangerous application configurations in Android. Figure 4.2 depicts our procedure, which consists of five main activities.

Step 1: Identify Assets: Instead of identifying assets from functional requirements, we extract them from the features on the Android platform. Google has identified many assets already in the form of permission labels protecting resources. Moreover, as the broadcasted Intent messages (e.g. those sent by the system) impact both platform and application operation, they are assets. Lastly, all components (Activities, etc.) of system applications are assets. While they are not necessarily protected by permission labels, many applications call upon them to operate.

As an example, Android defines the `RECORD_AUDIO` permission to protect its audio recorder. Here, we consider the asset to be microphone input, as it records the user's voice during phone conversations. Android also defines permissions for making phone calls and observing when the phone state changes. Hence, call activity is an asset.

Step 2: Identify Functional Requirements: Next, we carefully study each asset to specify corresponding functional descriptions. These descriptions indicate how the asset interacts with the rest of the phone and third-party applications. This step is vital to our design, because both assets and functional descriptions are necessary to investigate realistic threats.

Continuing the assets identified above, when the user receives an incoming call, the system broadcasts an Intent to the `PHONE_STATE` action string. It also notifies any applications that have registered a `PhoneStateListener` with the system. The same notifications are sent on outgoing call. Another Intent to the `NEW_OUTGOING_CALL` action string is also broadcasted. Furthermore, this additional broadcast uses the "ordered" option, which serializes the broadcast and allows any recipient to cancel it. If this occurs, subsequent Broadcast Receivers will not receive the Intent message. This feature allows, for example, an application to redirect international calls to the number for a calling card. Finally, audio can be recorded using the `MediaRecorder` API.

Step 3: Determine Assets Security Goals and Threats: In general, security requirements engineering considers high level security goals such as confidentiality, integrity, and availability. For each asset, we must determine which (if not all) goals are appropriate. Next, we consider how the functional requirements can be abused with respect to the remaining security goals. Abuse cases that violate the security goals provide *threat descriptions*. We use the malware motivations described in Section 2.1 to motivate our threats. Note that defining threat descriptions sometimes requires a level of creativity. However, trained security experts will find most threats straightforward after defining the functional requirements.

Continuing our example, we focus on the confidentiality of the microphone input and phone state notifications. These goals are abused if a malware records audio during voice call and transmits it over the Internet (i.e., premeditated spyware). The corresponding threat description becomes, "*spyware can breach the user's privacy by detecting the phone call activity, recording the conversation, and sending it to the adversary via the Internet.*"

Step 4: Develop Asset's Security Requirements: Next, we define security requirements from the threat descriptions. Recall from our earlier discussion, security requirements are constraints on functional requirements. That is, they specify who can exercise functionality or conditions under which functionality may occur. Frequently, this process consists of determining which sets of functionality are required to compromise a threat. The requirement is the security rule that restricts the ability for this functionality to be exercised in concert.

We observe that the eavesdropper requires a) notification of an incoming or outgoing call, b) the ability to record audio, and c) access to the Internet. Therefore, our security requirement, which acts as Kirin security rule, becomes, "*an application must not be able to receive phone state, record audio, and access the Internet.*"

Step 5: Determine Security Mechanism Limitations: Our final step caters to the practical limitations of our intended enforcement mechanism. Our goal is to identify potentially dangerous configurations at install time. Therefore, we cannot ensure runtime support beyond what Android already provides. Additionally, we are limited to the information available in an application package manifest. For both these reasons, we must refine our list of security requirements (i.e., Kirin security rules). Some rules may simply not be enforceable. For instance, we cannot ensure only a fixed number of SMS messages are sent during some time period [113], because Android does not support history-based policies. Security rules must also be translated to be expressed in terms of the security configuration available in the package manifest. This usually consists of identifying the permission labels used to protect functionality. Finally, as shown in Figure 4.2, the iteration between Steps 4 and 5 is required to adjust the rules to work within our limitations. Additionally, security rules can be subdivided to be more straightforward.

The permission labels corresponding to the restricted functionality in our running example include `READ_PHONE_STATE`, `PROCESS_OUTGOING_CALLS`, `RECORD_AUDIO`, and `INTERNET`. Furthermore, we subdivide our security rule to remove the disjunctive logic resulting from multiple ways for the eavesdropper to be notified of voice call activity. Hence, we create the following adjusted security rules: a) "*an application must not have the* `READ_PHONE_STATE`*,* `RECORD_AUDIO`*, and* `INTERNET` *permissions.*" and the nearly identical b) "*an application must not have the* `PROCESS_OUTGOING_CALLS`*,* `RECORD_AUDIO`*, and* `INTERNET` *permissions.*"

(1) *An application must not have the* `SET_DEBUG_APP` *permission label.*

(2) *An application must not have* `PHONE_STATE`*,* `RECORD_AUDIO`*, and* `INTERNET` *permission labels.*

(3) *An application must not have* `PROCESS_OUTGOING_CALL`*,* `RECORD_AUDIO`*, and* `INTERNET` *permission labels.*

(4) *An application must not have* `ACCESS_FINE_LOCATION`*,* `INTERNET`*, and* `RECEIVE_BOOT_COMPLETE` *permission labels.*

(5) *An application must not have* `ACCESS_COARSE_LOCATION`*,* `INTERNET`*, and* `RECEIVE_BOOT_COMPLETE` *permission labels.*

(6) *An application must not have* `RECEIVE_SMS` *and* `WRITE_SMS` *permission labels.*

(7) *An application must not have* `SEND_SMS` *and* `WRITE_SMS` *permission labels.*

(8) *An application must not have* `INSTALL_SHORTCUT` *and* `UNINSTALL_SHORTCUT` *permission labels.*

(9) *An application must not have the* `SET_PREFERRED_APPLICATION` *permission label and receive Intents for the* `CALL` *action string.*

Figure 4.3. Sample Kirin security rules to mitigate malware

4.3.2 Sample Malware Mitigation Rules

The remainder of this section discusses Kirin security rules we developed following our 5-step methodology. For readability and ease of exposition, we have enumerated the precise security rules in Figure 4.3. We refer to the rules by the indicated numbers for the remainder of the paper. We loosely categorize Kirin security rules by their complexity.

4.3.2.1 Single Permission Security Rules

Recall that a number of Android's "dangerous" permissions may be "too dangerous" for some production environments. We discovered several such permission labels. For instance, the `SET_DEBUG_APP` permission "...allows an application to turn on debugging for another application." (according to available documentation). The corresponding API is "hidden" in the most recent SDK environment (at the time of writing, version 1.1r1). The hidden APIs are not accessible by third-party applications but only by system applications. However, hidden APIs are no substitute for security. A malware author can simply download Android's source code and build an SDK that includes the API. The malware then, for instance, can disable anti-virus software. Rule 1 ensures third party applications do not have the `SET_DEBUG_APP` permission. Similar rules can be made for other permission labels protecting hidden APIs (e.g., Bluetooth APIs not yet considered mature enough for general use).

4.3.2.2 Multiple Permission Security Rules

Voice and location eavesdropping malware requires permissions to record audio and access location information. However, legitimate applications use these permissions as well. Therefore, we must define rules with respect to multiple permissions. To do this, we consider the minimal set of functionality required to compromise a threat. Rules 2 and 3 protect against the voice call eavesdropper used as a running example in Section 4.3.1. Similarly, Rules 4 and 5 protect against a location tracker. In this case, the malware starts executing on boot. In these security rules, we assume the malware starts on boot by defining a Broadcast Receiver to receive the `BOOT_COMPLETE` action string. Note that the `RECEIVE_BOOT_COMPLETE` permission label protecting this broadcast is a "normal" permission (and hence is always granted). However, the permission label provides valuable insight into the functional requirements of an application. In general, *Kirin security rules are more expressible as the number of available permission labels increases.*

Rules 6 and 7 consider malware's interaction with SMS. Rule 6 protects against malware hiding or otherwise tampering with incoming SMS messages. For example, SMS can be used as a control channel for the malware. However, the malware author does not want to alert the user, therefore immediately after an SMS is received from a specific sender, the SMS Content Provider is modified. In practice, we found that our sample malware could not remove the SMS notification from the phone's status bar. However, we were able to modify the contents of the SMS message in the Content Provider. While we could not hide the control message completely, we were able to change the message to appear as spam. Alternatively, a similar attack could ensure the user never receives SMS messages from a specific sender, for instance PayPal or a financial institution. Such services often provide out-of-band transaction confirmations. Blocking an SMS message from this sender could hide other activity performed by the malware. While this attack is also limited by notifications in the status bar, again, the message contents can be transformed as spam.

Rule 7 mitigates mobile bots sending SMS spam. Similar to Rule 6, this rule ensures the malware cannot remove traces of its activity. While Rule 7 does not prevent the SMS spam messages from being sent, it increases the probability that the user becomes aware of the activity.

Finally, Rule 8 makes use of the duality of some permission labels. Android defines separate permissions for installing and uninstalling shortcuts on the phone's home screen. This rule ensures that a third-party application cannot have both. If an application has both, it can redirect the shortcuts for frequently used applications to a malicious one.

For instance, the shortcut for the web browser could be redirected to an identically appearing application that harvests passwords.

4.3.2.3 Permission and Interface Security Rules

Permissions alone are not always enough to characterize malware behavior. Rule 9 provides an example of a rule considering both a permission and an action string. This specific rule prevents malware from replacing the default voice call dialer application without the user's knowledge. Normally, if Android detects two or more applications contain Activities to handle an Intent message, the user is prompted which application to use. This interface also allows the user to set the current selection as default. However, if an application has the `SET_PREFERRED_APPLICATION` permission label, it can set the default without the user's knowledge. Google marks this permission as "dangerous"; however, users may not fully understand the security implications of granting it. Rule 9 combines this permission with the existence of an Intent filter receiving the `CALL` action string. Hence, we can allow a third-party application to obtain the permission as long as it does not also handle voice calls. Similar rules can be constructed for other action strings handled by the trusted computing base.

4.4 Kirin Security Language

We now describe the Kirin Security Language (KSL) to encode security rules for the Kirin security service. Kirin uses an application's package manifest as input. The rules identified in Section 4.3 only require knowledge of the permission labels requested by an application and the action strings used in Intent filters. This section defines the KSL syntax and formally defines its semantics.

4.4.1 KSL Syntax

Figure 4.4 defines the Kirin Security Language in BNF notation. A KSL rule-set consists of a list of rules. A rule indicates combinations of permission labels and action strings that should not be used by third-party applications. Each rule begins with the keyword "`restrict`". The remainder of the rule is the conjunction of sets of permissions and action strings received. Each set is denoted as either "`permission`" or "`receive`", respectively.

$$\langle\text{rule-set}\rangle ::= \langle\text{rule}\rangle \mid \langle\text{rule}\rangle\ \langle\text{rule-set}\rangle \tag{4.1}$$

$$\langle\text{rule}\rangle ::= \text{``}\texttt{restrict}\text{''}\ \langle\text{restrict-list}\rangle \tag{4.2}$$

$$\langle\text{restrict-list}\rangle ::= \langle\text{restrict}\rangle \mid \langle\text{restrict}\rangle\ \text{``}\texttt{and}\text{''}\ \langle\text{restrict-list}\rangle \tag{4.3}$$

$$\begin{aligned}\langle\text{restrict}\rangle ::= {} & \text{``}\texttt{permission [}\text{''}\ \langle\text{const-list}\rangle\ \text{``}\texttt{]}\text{''} \mid \\ & \text{``}\texttt{receive [}\text{''}\ \langle\text{const-list}\rangle\ \text{``}\texttt{]}\text{''}\end{aligned} \tag{4.4}$$

$$\langle\text{const-list}\rangle ::= \langle\text{const}\rangle \mid \langle\text{const}\rangle\ \text{``}\texttt{,}\text{''}\ \langle\text{const-list}\rangle \tag{4.5}$$

$$\langle\text{const}\rangle ::= \text{``}\texttt{'}\text{''}\texttt{[A-Za-z0-9\_.]+}\text{``}\texttt{'}\text{''} \tag{4.6}$$

Figure 4.4. KSL syntax in BNF.

4.4.2 KSL Semantics

We now define a simple logic to represent a set of rules written in KSL. Let $\mathcal{R}$ be set of all rules expressible in KSL. Let $\mathcal{P}$ be the set of possible permission labels and $\mathcal{A}$ be the set of possible action strings used by Activities, Services, and Broadcast Receivers to receive Intents. Then, each rule $r_i \in \mathcal{R}$ is a tuple $(2^{\mathcal{P}}, 2^{\mathcal{A}})$.[2] We use the notation $r_i = (P_i, A_i)$ to refer to a specific subset of permission labels and action strings for rule r_i, where $P_i \in 2^{\mathcal{P}}$ and $A_i \in 2^{\mathcal{A}}$.

Let $R \subseteq \mathcal{R}$ correspond to a set of KSL rules. We construct R from the KSL rules as follows. For each $\langle\text{rule}\rangle_i$, let P_i be the union of all sets of "`permission`" restrictions, and let A_i be the union of all sets of "`receive`" restrictions. Then, create $r_i = (P_i, A_i)$ and place it in R. The set R directly corresponds to the set of KSL rules and can be formed in time linear to the size of the KSL rule set (proof by inspection).

Next we define a configuration based on package manifest contents. Let $\mathcal{C}$ be the set of all possible configurations extracted from a package manifest. We need only capture the set of permission labels used by the application and the action strings used by its Activities, Services, and Broadcast Receivers. Note that the package manifest does not specify action strings used by dynamic Broadcast Receivers; however, we use this fact to our advantage (as discussed in Section 5.7). We define configuration $c \in \mathcal{C}$ as a tuple $(2^{\mathcal{P}}, 2^{\mathcal{A}})$. We use the notation $c_t = (P_t, A_t)$ to refer to a specific subset of permission labels and action strings used by a target application t, where $P_t \in 2^{\mathcal{P}}$ and $A_t \in 2^{\mathcal{A}}$.

We now define the semantics of a set of KSL rules. Let $fail : \mathcal{C} \times \mathcal{R} \to \{\texttt{true}, \texttt{false}\}$ be a function to test if an application configuration fails a KSL rule. Let c_t be the

---

[2] We use the standard notation 2^X represent the power set of a set X, which is the set of all subsets including $\emptyset$.

configuration for target application t and r_i be a rule. Then, we define $fail(c_t, r_i)$ as:

$$(P_t, A_t) = c_t, (P_i, A_i) = r_i, P_i \subseteq P_t \wedge A_i \subseteq A_t$$

Clearly, $fail(\cdot)$ operates in time linear to the input, as a hash table can provide constant time set membership checks.

Let $F_R : \mathcal{C} \to \mathcal{R}$ be a function returning the set of all rules in $R \in 2^{\mathcal{R}}$ for which an application configuration fails:

$$F_R(c_t) = \{r_i | r_i \in R, fail(c_t, r_i)\}$$

Then, we say the configuration c_t passes a given KSL rule-set R if $F_R(c_t) = \emptyset$. Note that $F_R(c_t)$ operates in time linear to the size of c_t and R. Finally, the set $F_R(c_t)$ can be returned to the application installer to indicate which rules failed. This information facilitates the optional user override extension described in Section 4.2.

4.5 Kirin Security Service

For flexibility, Kirin is designed as a security service running on the mobile phone. The existing software installer interfaces directly with the security service. This approach follows Android's design principle of allowing applications to be replaced based on manufacturer and consumer interests. More specifically, a new installer can also use Kirin.

We implemented Kirin as an Android application. The primary functionality exists within a Service component that exports an RPC interface used by the software installer. This service reads KSL rules from a configuration file. At install time, the installer passes the file path to the package archive (.apk file) to the RPC interface. Then, Kirin parses the package to extract the security configuration stored in the package manifest. The `PackageManager` and `PackageParser` APIs provide the necessary information. The configuration is then evaluated against the KSL rules. Finally, the passed/failed result is returned to the installer with the list of the violated rules. Note that Kirin service does not access any critical resources of the platform hence does not require any permissions.

4.6 Evaluation

Practical security rules must both mitigate malware and allow legitimate applications to be installed. Section 4.3 argued that our sample security rules can detect specific

Table 4.1. Applications failing Rule 2

Application	**Description**
Walki Talkie Push to Talk	Walkie-Talkie style voice communication.
Shazam	Utility to identify music tracks.
Inauguration Report	Collaborative journalism application.

types of malware. However, Kirin's certification technique conservatively detects dangerous functionality, and may reject legitimate applications. In this section, we evaluate our sample security rules against real applications from the Android Market. While the Android Market does not perform rigorous certification, we initially assume it does not contain malware. Any application not passing a security rule requires further investigation. Overall, we found very few applications where this was the case. On one occasion, we found a rule could be refined to reduce this number further.

Our sample set consisted of a snapshot of a subset of popular applications available in the Android Market in late January 2009. We downloaded the top 20 applications from each of the 16 categories, producing a total of 311 applications (one category only had 11 applications). We used Kirin to extract the appropriate information from each package manifest and ran the $F_R(\cdot)$ algorithm described in Section 4.4.

4.6.1 Empirical Results

Our analysis tested all 311 applications against the security rules listed in Figure 4.3. Of the 311 applications, only 12 failed to pass all 9 security rules. Of these, 3 applications failed Rule 2 and 9 applications failed Rules 4 and 5. These failure sets were disjoint, and no applications failed the other six rules.

Table 4.1 lists the applications that fail Rule 2. Recall that Rule 2 defends against a malicious eavesdropper by failing any application that can read phone state, record audio, and access the Internet. However, none of the applications listed in Table 4.1 exhibit eavesdropper-like characteristics. Considering the purpose of each application, it is clear why they require the ability to record audio and access the Internet. We initially speculated that the applications stop recording upon an incoming call. However, this was not the case. We disproved our speculation for Shazam and Inauguration Report and were unable to determine a solid reason for the permission label's existence, as no source code was available.

After realizing that simultaneous access to phone state and audio recording is in fact beneficial (i.e., to stop recording on incoming call), we decided to refine Rule 2. Our goal is to protect against an eavesdropper that automatically records a voice call on either incoming or outgoing call. Recall that there are two ways to obtain the phone state: 1) register a Broadcast Receiver for the `PHONE_STATE` action string, and 2) register a `PhoneStateListener` with the system. If a static Broadcast Receiver is used for the former case, the application is automatically started on incoming and outgoing call. The latter case requires the application to be already started, e.g., by the user, or on boot. We need only consider cases where it is started automatically. Using this information, we split Rule 2 into two new security rules. Each appends an additional condition. The first appends a restriction on receiving the `PHONE_STATE` action string. Note that since Kirin only uses Broadcast Receivers defined in the package manifest, we will not detect dynamic Broadcast Receivers that cannot be used to automatically start the application. The second rule appends the boot complete permission label used for Rule 4. Rerunning the applications against our new set of security rules, we found that only the Walkie Talkie application failed our rules, thus reducing the number of failed applications to 10.

Table 4.2 lists the applications that fail Rules 4 and 5. Recall that these security rules detect applications that start on boot and access location information and the Internet. The goal of these rules is to prevent location tracking software. Of the nine applications listed in Table 4.2, the first five provide functionality that directly contrast with the rule's goal. In fact, Kirin correctly identified both AccuTracking and GPS Tracker as dangerous. Both Loopt and Twidroid are popular social networking applications; however, they do in fact provide potentially dangerous functionality, as they can be configured to automatically start on boot without the user's knowledge. Finally, Pintail is designed to report the phone's location in response to an SMS message with the correct password. While this may be initiated by the user, it may also be used by an adversary to track the user. Again, Kirin correctly identified potentially dangerous functionality.

The remaining four applications in Table 4.2 result from the limitations in Kirin's input. That is, Kirin cannot inspect how an application uses information. In the previous cases, the location information was used to track the user. However, for these applications, the location information is used to supplement Internet data retrieval. Both WeatherBug and Homes use the phone's location to filter information from a website. Additionally, there is little correlation between location and the ability to start on boot. On the other hand, the T-Mobile Hotspot WiFi finder provides useful functionality by starting on boot and notifying the user when the phone is near such wireless networks. However, in all three of these cases, we do not believe access to "fine" location is re-

Table 4.2. Applications failing Rule 4 and 5

Application	**Description**
AccuTracking	Client for real-time GPS tracking service (AccuTracking).
GPS Tracker*	Client for real-time GPS tracking service (InstaMapper).
Loopt	Geosocial networking application that shares location with friends.
Twidroid	Twitter client that optionally allows automatic location tweets.
Pintail	Reports the phone location in response to SMS message.
WeatherBug	Weather application with automatic weather alerts.
Homes	Classifieds application to aid in buying or renting houses.
T-Mobile Hotspot	Utility to discover nearby nearby T-Mobile WiFi hotspots.
Power Manager	Utility to automatically manage radios and screen brightness.

* Did not fail Rule 5

quired; location with respect to a cellular tower is enough to determine a city or even a city block. Removing this permission would allow the applications to pass Rule 4. Finally, we were unable to determine why Power Manager required location information. We initially thought it switched power profiles based on location, but did not find an option.

In summary, 12 of the 311 applications did not pass our initial security rules. We reduced this to 10 after revisiting our security requirements engineering process to better specify the rules. This is the nature of security requirements engineering, which an ongoing process of discovery. Of the remaining 10, Kirin correctly identified potentially dangerous functionality in 5 of the applications, which should be installed with extreme caution. The remaining five applications assert a dangerous configuration of permissions, but were used within reasonable functional needs based on application descriptions. Therefore, Kirin's conservative certification technique only requires user involvement for approximately 1.6% of applications (according to our sample set). From this, we observe that Kirin can be very effective at practically mitigating malware.

4.6.2 Mitigating Malware

We have shown that Kirin can practically mitigate certain types of malware. However, Kirin is not a complete solution for malware protection. We constructed practical security by considering different malicious motivations. Some motivations are more difficult to practically detect with Kirin. Malware of destructive or proof-of-concept origins may only require one permission label to carry out its goals. For example, malware might intend to remove all contacts from the phone's address book. Kirin cannot simply deny all third-party applications the ability to write to the address book. Such a rule would fail for an application that merges Web-based address books (e.g., Facebook).

Kirin is more valuable in defending against complex attacks requiring multiple functionalities. We discussed a number of rules that defend against premeditated spyware. Rule 8 defends against shortcut replacement, which can be used by information scavengers to trick the user into using a malicious Web browser. Furthermore, Rule 6 can help hide financial transactions that might result from obtained usernames and passwords. Kirin can also help mitigate the effects of botnets. For example, Rule 7 does not let an application hide outbound SMS spam. This requirement can also be used to help a user become aware of SMS sent to premium numbers (i.e., direct payoff malware). However, Kirin could be more effective if Android's permission labels distinguished between sending SMS messages to contacts in the address book verses arbitrary numbers.

Kirin's usefulness to defend against ad-ware is unclear. Many applications are supported by advertisements. However, applications that continually pester the user are undesirable. Android does not define permissions to protect notification mechanisms (e.g., the status bar), but even with such permissions, there are many legitimate reasons for using notifications. Despite this, in best case, the user can identify the offending application and uninstall it.

Finally, Kirin's expressibility is restricted by the policy that Android enforces. Android policy itself is static and does not support runtime logic. Therefore, it cannot enforce that no more than 10 SMS messages are sent per hour [113]. However, this is a limitation of Android and not Kirin.

4.7 Discovered Vulnerabilities

The process of retrofitting security requirements for Android had secondary effects. In addition to identifying rules for Kirin, we discovered a number of configuration and implementation flaws. Step 1 identifies assets. However, not all assets are protected by permissions. In particular, in early versions of our analysis discovered that the Intent message broadcasted by the system to the `SMS_RECEIVED` action string was not protected. Hence, any application can create a forged SMS that appears to have come from the cellular network. Upon notifying Google of the problem, the new permission `BROADCAST_SMS_RECEIVED` has been created and protects the system broadcast as of Android version 1.1. We also discovered an unprotected Activity component in the phone application that allows a malicious application to make phone calls without having the `CALL_PHONE` permission. This configuration flaw has also been fixed. As we continued our investigation with the most recent version of Android (v1.1r1), we discovered a number of APIs do not check permissions as prescribed in the documentation. All of these flaws show the value in defining security requirements. Kirin relies on Android to enforce security at runtime. Ensuring the security of a phone requires a complete solution, of which Kirin is only part.

4.8 Summary

As users continue to become comfortable downloading software for smartphones, malware targeting phones will increase. Kirin provides lightweight certification at install time that does not require burdensome code inspection for each application. We have shown that Kirin can express meaningful security rules to mitigate malware. Kirin's conservative certification technique is appropriate to detect many types of dangerous functionality. However, dangerous behavior that requires configuration identical to desired benign behavior is inappropriate to detect with configuration-level analysis. For example, simultaneous access to location information and the Internet. While we showed non-security configuration artifacts can help distinguish benign and malicious behaviors, more sophisticated analysis techniques are needed. In the next chapter, we consider in greater detail the general class of information misuse.

Chapter 5

Dynamic Tracking for Realtime Privacy Monitoring on Smartphones

Misuse of privacy sensitive information has become a high-profile risk for smartphones. Several academic studies [6, 32] have identified misuse by popular applications, including work [6] contributing to this chapter. Lack of informed consent for use of privacy values, such as phone identifiers, has also resulted in lawsuits [199]. This high-profile attention motivates the creation to technology to identify and mitigate such privacy risks.

Privacy sensitive values such as location are difficult to protect in the smartphone environment, because they often transmitted to the Internet for legitimate purposes. Hence, analysis of security policy is insufficient. To understand how such values are used, one must look inside applications. Therefore, this chapter asks the question, *what do applications do with permissions?* In particular, this chapter uses dynamic taint analysis to focus on permissions that grant access to sensitive values.

5.1 Identifying Privacy Risks in Smartphone Applications

Resolving the tension between the utility of running third-party mobile applications and the privacy risks they pose is a critical challenge for smartphone platforms. Smartphone operating systems currently provide only coarse-grained controls for regulating whether an application can *access* private information, but provide little insight into how private information is actually used. For example, if a user allows an application to access her location information, she has no way of knowing if the application will send her location to a location-based service, to advertisers, to the application developer, or to any other entity. As a result, users must blindly trust that applications will properly handle their private data.

This chapter describes *TaintDroid*, an extension to the Android platform that tracks the flow of privacy sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors–in realtime–how these applications access and manipulate users' personal data. Our primary goals are to detect when sensitive data leaves the system via untrusted applications and to facilitate analysis of applications by phone users or external security services [200, 201].

Analysis of applications' behavior requires sufficient contextual information about what data leaves a device and where it is sent. Thus, TaintDroid automatically labels (taints) data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. When tainted data are transmitted over the network, or otherwise leave the system, TaintDroid logs the data's labels, the application responsible for transmitting the data, and the data's destination. Such realtime feedback gives users and security services greater insight into what mobile applications are doing, and can potentially identify misbehaving applications.

To be practical, the performance overhead of the TaintDroid runtime must be minimal. Unlike existing solutions that rely on heavy-weight whole-system emulation [155, 148], we leveraged Android's virtualized architecture to integrate four granularities of taint propagation: variable-level, method-level, message-level, and file-level. Though the individual techniques are not new, our contributions lie in the integration of these techniques and in identifying an appropriate trade-off between performance and accuracy for resource constrained smartphones. Experiments with our prototype for Android show that tracking incurs a runtime overhead of less than 14% for a CPU-bound microbenchmark. More importantly, interactive third-party applications can be monitored with negligible perceived latency.

We evaluated the accuracy of TaintDroid using 30 randomly selected, popular Android applications that use location, camera, or microphone data. TaintDroid correctly flagged 105 instances in which these applications transmitted tainted data; of the 105, we determined that 37 were clearly legitimate. TaintDroid also revealed that 15 of the 30 applications reported users' locations to remote advertising servers. Seven applications collected the device ID and, in some cases, the phone number and the SIM card serial number. In all, two-thirds of the applications in our study used sensitive data suspiciously. Our findings demonstrate that TaintDroid can help expose potential misbehavior by third-party applications.

Like similar information-flow tracking systems [155, 148], a fundamental limitation of TaintDroid is that it can be circumvented through leaks via implicit flows. The use of implicit flows to avoid taint detection is, in and of itself, an indicator of malicious intent, and may well be detectable through other techniques such as automated static code analysis [144, 139] as we discuss in Section 5.8.

The remainder of this chapter is organized as follows: Section 5.2 provides a high-level overview of TaintDroid, Section 5.3 describes detailed background information on the Android platform specific to TaintDroid, Section 5.4 describes our TaintDroid design, Section 5.5 describes the taint sources tracked by TaintDroid, Section 5.6 presents results from our Android application study, Section 5.7 characterizes the performance of our prototype implementation, Section 5.8 discusses the limitations of our approach, and Section 5.9 summarizes our conclusions.

5.2 Approach Overview

We seek to design a framework that allows users to monitor how third-party smartphone applications handle their private data in realtime. Many smartphone applications are closed-source, therefore, static source code analysis is infeasible. Even if source code is available, runtime events and configuration often dictate information use; realtime monitoring accounts for these environment specific dependencies.

Monitoring network disclosure of privacy sensitive information on smartphones presents several challenges:

- *Smartphones are resource constrained.* The resource limitations of smartphones precludes the use of heavyweight information tracking systems such as Panorama [148].
- *Third-party applications are entrusted with several types of privacy sensitive information.* The monitoring system must distinguish multiple information types, which requires additional computation and storage.
- *Context-based privacy sensitive information is dynamic and can be difficult to identify even when sent in the clear.* For example, geographic locations are pairs of floating point numbers that frequently change and are hard to predict.
- *Applications can share information.* Limiting the monitoring system to a single application does not account for flows via files and IPC between applications,

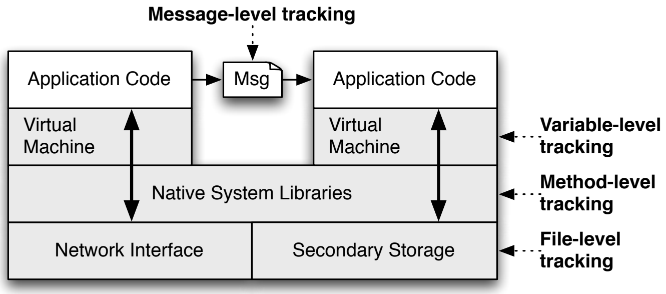

Figure 5.1. Multi-level approach for performance efficient taint tracking within a common smartphone architecture.

including core system applications designed to disseminate privacy sensitive information.

We use dynamic taint analysis [148, 146, 147, 150, 164] (also called "taint tracking") to monitor privacy sensitive information on smartphones. Sensitive information is first identified at a *taint source*, where a *taint marking* indicating the information type is assigned. Dynamic taint analysis tracks how labeled data impacts other data in a way that might leak the original sensitive information. This tracking is often performed at the instruction level. Finally, the impacted data is identified before it leaves the system at a *taint sink* (usually the network interface).

Existing taint tracking approaches have several limitations. First and foremost, approaches that rely on instruction-level dynamic taint analysis using whole system emulation [148, 155, 157] incur high performance penalties. Instruction-level instrumentation incurs 2-20 times slowdown [148, 155] in addition to the slowdown introduced by emulation, which is not suitable for realtime analysis. Second, developing accurate taint propagation logic has proven challenging for the x86 instruction set [202, 203]. Implementations of instruction-level tracking can experience taint explosion if the stack pointer becomes falsely tainted [204] and taint loss if complicated instructions such as CMPXCHG, REP MOV are not instrumented properly [150]. While most smartphones use the ARM instruction set, similar false positives and false negatives could arise.

Figure 5.1 presents our approach to taint tracking on smartphones. We leverage architectural features of virtual machine-based smartphones (e.g., Android, BlackBerry, and J2ME-based phones) to enable efficient, system-wide taint tracking using fine-grained labels with clear semantics. First, we instrument the VM interpreter to provide *variable-level tracking* within untrusted application code.[1] Using variable semantics provided by

---

[1] A similar approach can be applied to just-in-time compilation by inserting tracking code within the generated binary.

the interpreter provides valuable context for avoiding the taint explosion observed in the x86 instruction set. Additionally, by tracking variables, we maintain taint markings only for data and not code. Second, we use *message-level tracking* between applications. Tracking taint on messages instead of data within messages minimizes IPC overhead while extending the analysis system-wide. Third, for system-provided native libraries, we use *method-level tracking*. Here, we run native code without instrumentation and patch the taint propagation on return. These methods accompany the system and have known information flow semantics. Finally, we use *file-level* tracking to ensure persistent information conservatively retains its taint markings.

To assign labels, we take advantage of the well-defined interfaces through which applications access sensitive data. For example, all information retrieved from GPS hardware is location-sensitive, and all information retrieved from an address book database is contact-sensitive. This avoids relying on heuristics [205] or manual specification [150] for labels. We expand on information sources in Section 5.5.

In order to achieve this tracking at multiple granularities, our approach relies on the firmware's integrity. The taint tracking system's trusted computing base includes the virtual machine executing in userspace and any native system libraries loaded by the untrusted interpreted application. However, this code is part of the firmware, and is therefore trusted. Applications can only escape the virtual machine by executing native methods. In our target platform (Android), we modified the native library loader to ensure that applications can only load native libraries from the firmware and not those downloaded by the application. Note that an early 2010 survey of the top 50 most popular free applications in each category of the Android Market [206] (1100 applications in total) revealed that less than 4% included a `.so` file. A similar survey conducted in mid 2010 revealed this fraction increased to 5%, which indicates there is growth in the number of applications using native third-party libraries, but that the number of affected applications remains small.

In summary, we provide a novel, efficient, system-wide, multiple-marking, taint tracking design by combining multiple granularities of information tracking. While some techniques such as variable tracking within an interpreter have been previously proposed (see Section 3.3), to our knowledge, our approach is the first to extend such tracking system-wide. By choosing a multiple granularity approach, we balance performance and accuracy. As we show in Sections 5.6 and 5.7, our system-wide approach is both highly efficient (~14% CPU overhead and ~4.4% memory overhead for simultaneously tracking 32 taint markings per data unit) and accurately detects many suspicious network packets.

5.3 Information Processing in Android

Android [207] is a Linux-based, open source, mobile phone platform. Most core phone functionality is implemented as applications running on top of a customized middleware. The middleware itself is written in Java and C/C++. Applications are written in Java and compiled to a custom byte-code known as the Dalvik EXecutable (DEX) byte-code format. Each application executes within its Dalvik VM interpreter instance. Each instance executes as unique UNIX user identities to isolate applications within the Linux platform subsystem. Applications communicate via the binder IPC mechanism. Binder provides transparent message passing based on parcels. We now discuss topics necessary to understand our tracking system.

Dalvik VM Interpreter: DEX is a register-based machine language, as opposed to Java byte-code, which is stack-based. Each DEX method has its own predefined number of virtual registers (which we frequently refer to as simply "registers"). The Dalvik VM interpreter manages method registers with an internal execution state stack; the current method's registers are always on the top stack frame. These registers loosely correspond to local variables in the Java method and store primitive types and object references. All computation occurs on registers, therefore values must be loaded from and stored to class fields before use and after use. Note that DEX uses class fields for all long term storage, unlike hardware register-based machine languages (e.g., x86), which store values in arbitrary memory locations.

Native Methods: The Android middleware provides access to native libraries for performance optimization and third-party libraries such as OpenGL and Webkit. Android also uses Apache Harmony Java [208], which frequently uses system libraries (e.g., math routines). Native methods are written in C/C++ and expose functionality provided by the underlying Linux kernel and services. They can also access Java internals, and hence are included in our trusted computing base (see Section 5.2).

Android contains two types of native methods: internal VM methods and JNI methods. The internal VM methods access interpreter-specific structures and APIs. JNI methods conform to Java native interface standards specifications [209], which requires Dalvik to separate Java arguments into variables using a JNI call bridge. Conversely, internal VM methods must manually parse arguments from the interpreter's byte array

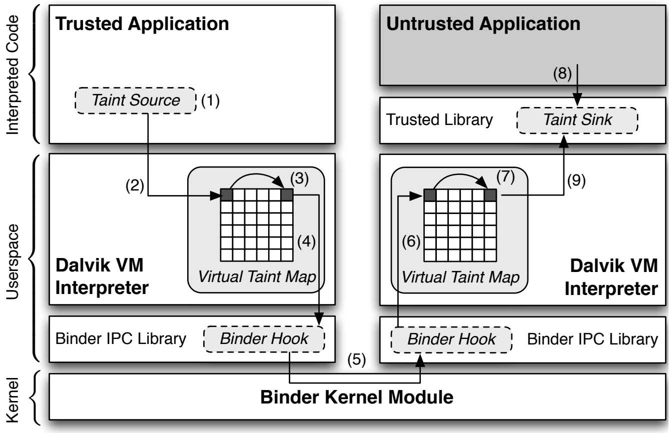

Figure 5.2. TaintDroid architecture within Android.

of arguments.

Binder IPC: All Android IPC occurs through binder. Binder is a component-based processing and IPC framework designed for BeOS, extended by Palm Inc., and customized for Android by Google. Fundamental to binder are *parcels*, which serialize both active and standard data objects. The former includes references to binder objects, which allows the framework to manage shared data objects between processes. A binder kernel module passes parcel messages between processes.

5.4 TaintDroid

TaintDroid is a realization of our multiple granularity taint tracking approach within Android. TaintDroid uses variable-level tracking within the VM interpreter. Multiple taint markings are stored as one *taint tag*. When applications execute native methods, variable taint tags are patched on return. Finally, taint tags are assigned to parcels and propagated through binder.

Figure 5.2 depicts TaintDroid's architecture. Information is tainted (1) in a trusted application with sufficient context (e.g., the location provider). The taint interface invokes a native method (2) that interfaces with the Dalvik VM interpreter, storing specified taint markings in the virtual taint map. The Dalvik VM propagates taint tags (3) according to data flow rules as the trusted application uses the tainted information. Every interpreter instance simultaneously propagates taint tags. When the trusted application uses the tainted information in an IPC transaction, the modified binder library (4) ensures the parcel has a taint tag reflecting the combined taint markings of all contained data. The parcel is passed transparently through the kernel (5) and received by the remote untrusted application. Note that only the interpreted code is untrusted. The modified binder library retrieves the taint tag from the parcel and assigns it to all values read from it (6). The remote Dalvik VM instance propagates taint tags (7) identically for the untrusted application. When the untrusted application invokes a library specified as a taint sink (8), e.g., network send, the library retrieves the taint tag for the data in question (9) and reports the event.

Implementing this architecture requires addressing several system challenges, including: *a*) taint tag storage, *b*) interpreted code taint propagation, *c*) native code taint propagation, *d*) IPC taint propagation, and *e*) secondary storage taint propagation. The remainder of this section describes our design.

5.4.1 Taint Tag Storage

The choice of how to store taint tags influences performance and memory overhead. Dynamic taint tracking systems commonly store tags for every data byte or word [148, 155]. Tracked memory is unstructured and without content semantics. Frequently taint tags are stored in non-adjacent shadow memory [148] and tag maps [150]. TaintDroid uses variable semantics within the Dalvik interpreter. We store taint tags adjacent to variables in memory, providing spatial locality.

Dalvik has five variable types that require taint storage: method local variables, method arguments, class static fields, class instance fields, and arrays. In all cases, we store a 32-bit bitvector with each variable to encode the taint tag, allowing 32 different taint markings.

Method Local Variables: Dalvik loads local variables into registers for use in methods. Registers contain primitive type values and object references, and are always 32 bits, with *long* and *double* type variables occupying two adjacent registers. The interpreter stores registers on an internal execution state stack. On method invocation, Dalvik pushes a new stack frame, allocating space for its registers. During execution, registers are referenced by an index offset from the current frame pointer. For example, register $v0$ is $fp[0]$, register $v1$ is $fp[1]$, and so on. On method termination, the stack frame is popped, losing the register values.

TaintDroid stores taint tags for each register (regardless of its current taint state) by allocating room for double the number of registers during the stack frame push. Taint tags are stored immediately after registers for efficient reference (as depicted in

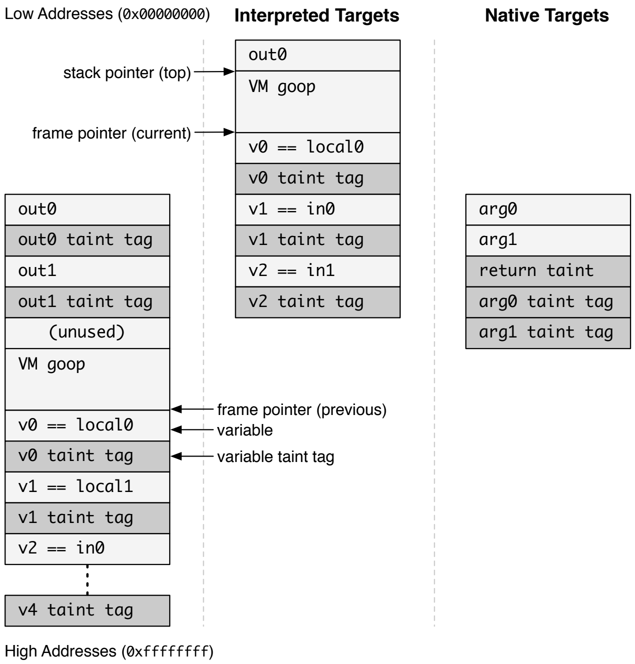

Figure 5.3. Modified Stack Format. Taint tags are interleaved between registers for interpreted method targets and appended for native methods. Dark grayed boxes represent taint tags.

Figure 5.3). TaintDroid accounts for tag storage by adjusting the frame pointer index for each register vi to $fp[2\cdot i]$ (a left bit shift), with the corresponding taint tag in $fp[2\cdot i+1]$.

Method Arguments: A target method is either interpreted or native. The Dalvik VM uses the execution state stack to pass arguments to both target types. Before a method is invoked, copies of the specified argument registers are pushed onto the stack (the copies disappear on method termination, which impacts the taint library design, as discussed in Section 5.4.6). If the target method is interpreted, the new values become high numbered registers in the callee stack frame. That is, interpreted method arguments become local variable registers and hence require consistent taint instrumentation. If the target method is native, a pointer to the stack top is passed to the native method. The target native method receives a pointer to a byte array from which it must parse 32 and 64-bit values in accordance to its method signature.

Figure 5.3 depicts TaintDroid's method argument modifications for both interpreted and native methods. Arguments for interpreted methods have interleaved taint tags for consistency with local variable taint storage. Native methods, on the other hand, expect a specific format in the received byte array of arguments. Therefore, interleaving taint tags for native method arguments would require pervasive modification. Not all native methods require taint tags for correct taint propagation (see Section 5.4.3). By appending argument taint tags as shown in Figure 5.3, we maintain compatibility and reduce source code modifications.

Finally, as is discussed in Section 5.4.3, TaintDroid's native method instrumentation also requires transfer of the return value taint tag. We use the interpreter stack to communicate the return value taint tag to the interpreter. This modification results in an unused 32-bit spacer for interpreted methods. This communication option maintains native method interface compatibility, which simplifies instrumentation.

Class Fields: DEX byte-code maintains Java's class and object semantics. Java defines two types of class fields: static and instance. Static fields store one value per class definition and are shared across all class instances. Instance fields store a different value for each class instance.

Static field storage is straightforward, as values are stored directly in a data structure managed by the interpreter, hence allowing adjacent storage of taint tags. Instance fields require more careful instrumentation. The corresponding data structure does not store values, but rather a byte-offset into a data object instance. Here, we interleave taint tags with values in the class instance data object, causing the taint tag to always exist in the 32-bits following the value. [2]

Arrays: TaintDroid stores one taint tag per array, which incurs significantly less storage than storing one tag per value. Per-value taint tag storage would be severely inefficient for Java *String* objects, as each character would require its own tag and causes string manipulation to copy individual character taint tags.

Storing one taint tag per array may result in false positives during taint propagation. For example, if untainted variable u is stored into array A at index 0 ($A[0]$) and tainted variable t is stored into $A[1]$, then array A is tainted. Later, if variable v is assigned to $A[0]$, v will be tainted, even though u was untainted. Fortunately, Java frequently uses objects, and object references are infrequently tainted (see Section 5.4.2), therefore such false positives are intuitively minimized.

---

[2] Readers familiar with Android may recognize that the optimized DEX (ODEX) format hardcodes field byte-offsets in the byte-code. However, the target device conventionally creates ODEX files on-demand, allowing TaintDroid to ensure compatibility.

5.4.2 Interpreted Code Taint Propagation

Taint tracking granularity and flow semantics influence performance and accuracy. TaintDroid implements variable-level taint tracking within the Dalvik VM interpreter. Variables provide valuable semantics for taint propagation, distinguishing data pointers from scalar values. TaintDroid primarily tracks primitive type variables (e.g., *int*, *float*, etc); however, there are cases when object references must become tainted to ensure taint propagation operates correctly; this section addresses why these cases exist. However, first we present taint tracking in the Dalvik machine language as a formal logic.

5.4.2.1 Taint Propagation Logic

The Dalvik VM operates on the unique DEX machine language instruction set, therefore we must design an appropriate propagation logic. We use a data flow logic, as tracking implicit flows requires static analysis and causes significant performance overhead and overestimation in tracking [210] (see Section 5.8). We begin by defining taint markings, taint tags, variables, and taint propagation. We then present our logic rules for DEX.

Definition 1 (Universe of Taint Markings $\mathcal{L}$)**.** Let each taint marking be a label l. We assume a fixed set of taint markings in any particular system. Example privacy-based taint markings include location, phone number, and microphone input. We define the universe of taint markings $\mathcal{L}$ to be the set of taint markings considered relevant for an application of TaintDroid.

Definition 2 (Taint Tag)**.** A taint tag is a set of taint markings. A taint tag t is in the power set of $\mathcal{L}$, denoted $2^{\mathcal{L}}$, which includes $\emptyset$. Each variable has an associated tag that is dynamically updated based on logic rules.

Definition 3 (Variable)**.** A variable is an instance of one of the five variable types described in Section 5.4.1 (method local variable, method argument, class static field, class instance field, and array). Variable types have different representations. The local and argument variables correspond to virtual registers, denoted v_x. Class field variables are denoted as f_x to indicate a field variable with class index x. f_x alone indicates a static field. Instance fields require an instance object and are denoted $v_y(f_x)$, where v_y is the instance object reference variable. Finally, $v_x[\cdot]$ denotes an array, where v_x is an array object reference variable.

Definition 4 (Virtual taint map function $\tau(\cdot)$)**.** Let v be a variable. $\tau(v)$ returns the taint tag t for variable v. $\tau(v)$ can also be used to assign a taint tag to a variable.

Table 5.1. DEX Taint Propagation Logic. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter. A, B, and C are byte-code constants.

Op Format	**Op Semantics**	**Taint Propagation**	**Description**
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to $v_B \cup v_C$ taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A to array $\cup$ index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A to $f_C \cup$ obj. ref. taint

Retrieval and assignment is distinguished by the position of $\tau(\cdot)$ w.r.t. the $\leftarrow$ symbol. When $\tau(v)$ appears on the right hand side of $\leftarrow$, $\tau(v)$ retrieves the taint tag for v. When $\tau(v)$ appears on the left hand side, $\tau(v)$ assigns the taint tag for v. For example, $\tau(v_1) \leftarrow \tau(v_2)$ copies the taint tag from variable v_2 to v_1.

Definitions 1-4 provide the primitives required to define runtime taint propagation for Dalvik VM. Table 5.1 captures the propagation logic. The table enumerates abstracted versions of the byte-code instructions specified in the DEX documentation. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables, respectively, maintained within the interpreter. A, B, and C are constants in the byte-code.

The taint propagation logic uses conservative data flow semantics for constant, move, arithmetic, and logic instructions. Destination register values are always completely overwritten, therefore, the taint tag is set explicitly for each instruction. Constant values are considered untainted and therefore do not contribute to the taint tag of the destination register. The interpreter maintains "hidden registers" for return and exception values. These registers require taint tag storage and corresponding propagation logic. The arithmetic and logic operations include unary negation, binary arithmetic, bit shifts, and bitwise AND and OR (abstracted as $\otimes$ in the table). Finally, the DEX byte-code does not require idioms to clear values (e.g., "`xor eax, eax`" in x86), therefore no special handling is required.

Array instructions propagate taint tags to and from array objects (recall that we store one taint tag per array). The *aput-op* instruction taint logic unions the existing array taint tag with the taint tag of the variable to be stored. The *aget-op* instruction logic assigns the destination register the union of the array and index taint tags. Note that this is a deviation from data flow, but is commonly included in taint propagation logic (e.g., in Panorama [148]) to account for translation tables commonly used for character conversion.

DEX also defines several array-related instructions not included in Table 5.1 (for brevity). The *array-length* instruction returns the length of an array. Some taint propagation logics taint array length to aid direct control flow propagation (e.g., Vogt et al. [165]). We only consider data flow propagation, therefore we assign a taint tag of $\emptyset$. Next, the *new-array* and *fill-new-array* instructions allocate a new array with constant values, therefore we set the new array's taint tag to $\emptyset$. Finally, the *fill-data-array* copies values from the byte-code into an array. We assign a taint tag of $\emptyset$ to the array if none of the original array content remains.

The field *put* and *get* instructions have data flow semantics similar to the move instructions. This intuition holds for static fields and the instance field *put* instruction. However, after investigating Dalvik VM runtime behavior, we determined instance *get* instructions (*iget*) should include the object reference taint tag as described in the next section.

Finally, there are two miscellaneous DEX instruction types not included in Table 5.1 that do not propagate taint tags but require instrumentation. DEX defines a set of *cmp-X* instructions that perform a comparison between registers and assigns a destination register the value of 0, 1, or -1. As we only consider data flow, we assign the destination register a taint tag of $\emptyset$. Note that propagating taint to the destination register would provide context for direct control flow taint propagation if a "taint scope" were statically extracted from the DEX byte-code before execution. Lastly, the *instance-of* instruction corresponds to the *instanceof* operation in Java. We do not consider an object instance's type as ever containing a taint marking, therefore we set the destination register's taint tag to $\emptyset$.

5.4.2.2 Tainting Object References

The propagation rules in Table 5.1 are straightforward with two exceptions. First, taint propagation logics commonly include the taint tag of an array index during lookup to handle translation tables (e.g., ASCII/UNICODE or character case conversion). For

```
public static Integer valueOf(int i) {
  if (i < -128 || i > 127) {
    return new Integer(i); }
  return valueOfCache.CACHE [i+128];
}
static class valueOfCache {
  static final Integer[] CACHE = new Integer[256];
  static {
    for(int i=-128; i<=127; i++) {
      CACHE[i+128] = new Integer(i); } }
}
```

Figure 5.4. Excerpt from Android's Integer class illustrating the need for object reference taint propagation.

example, consider a translation table from lowercase to upper case characters: if a tainted value "a" is used as an array index, the resulting "A" value should be tainted even though the "A" value in the array is not. Hence, the taint logic for *aget-op* uses both the array and array index taint. Second, when the array contains object references (e.g., an *Integer* array), the index taint tag is propagated to the object reference and not the object value. Therefore, we include the object reference taint tag in the instance *get* (*iget-op*) rule.

The code listed in Figure 5.4 demonstrates a real instance of where object reference tainting is needed. Here, *valueOf()* returns an Integer object for a passed *int*. If the *int* argument is between -128 and 127, *valueOf()* returns reference to a statically defined Integer object. *valueOf()* is implicitly called for conversion to an object. Consider the following definition and use of a method *intProxy()*.

```
Object intProxy(int val) { return val; }
int out = (Integer) intProxy(tVal);
```

Consider the case where *tVal* is an *int* with value 1 and taint tag TAG. When *intProxy()* is passed *tVal*, TAG is propagated to *val*. When *intProxy()* returns *val*, it calls *Integer.valueOf()* to obtain an *Integer* instance corresponding to the scalar variable *val*. In this case, *Integer.valueOf()* returns a reference to the static Integer object with value 1. The *value* field (of the *Integer* class) in the object has taint tag of $\emptyset$; however, since the *aget-op* propagation rule includes the taint of the index register, the object reference has a taint tag of TAG. Therefore, only by including the object reference taint tag when the *value* field is read from the Integer (i.e., the *iget-op* propagation rule), will the correct taint tag of TAG be assigned to *out*.

5.4.3 Native Code Taint Propagation

Native code is unmonitored in TaintDroid. Ideally, we achieve the same propagation semantics as the interpreted counterpart. Hence, we define two *necessary postconditions* for accurate taint tracking in the Java-like environment: 1) all accessed external variables (i.e., class fields referenced by other methods) are assigned taint tags according to data flow rules; and 2) the return value is assigned a taint tag according to data flow rules. TaintDroid achieves these postconditions through an assortment of manual instrumentation, heuristics, and method profiles, depending on situational requirements.

5.4.3.1 Internal VM Methods

Internal VM method arguments include a pointer to an array of 4-byte values containing Java arguments and a pointer to a return value. The stack augmentation shown in Figure 5.3 provides access to taint tags for both Java arguments and the return value. We manually inspect and instrument Dalvik's internal VM methods for taint propagation. Only a subset of the internal VM methods require modification. For those that do, we manually acquire taint tags appended to the Java argument array and assign a taint tag to the memory slot reserved for the return value taint tag. We also modify the interpreter to copy this value to the internally managed return value taint tag after the method terminates.

We identified 185 internal VM methods in Android version 2.1. This list was further narrowed by considering method names and argument types. We manually inspected and instrumented the remaining methods (if necessary). For example, the *System.arraycopy()* native method copies the contents of one array object to another. Our instrumentation acquires the taint tag stored in the source array and assigns it to the destination array. Several native methods implementing Java reflection also required instrumentation.

5.4.3.2 JNI Methods

JNI methods are called by a call bridge, which is an internal VM method. The call bridge parses the argument array based on a method descriptor string indicating the number and type of Java arguments. The call bridge then copies the values into the native instruction set calling convention. As a consequence, JNI methods cannot retrieve and return taint tags in the same way performed for internal VM methods. However, the call bridge provides a valuable mediation hook for generic and extensible taint propagation rules. We use a combination of heuristics and method profiles to capture information flow in JNI methods. The heuristic exists to minimize the effort required to define method profiles and is unnecessary, given an automated means of defining the profiles (as described below).

The propagation heuristic provides conservative propagation for JNI methods that only operate on primitive type variables (a common property). The heuristic calculates the union of the taint tags associated with the method arguments and assigns the result to the taint tag of the return value. For example, the *cos()* math library (a JNI method in Android) takes one argument and returns the cosine of that value, where there is a flow from the argument to the return value. Note this conservative calculation may cause false positives.

The heuristic has only false negatives for methods using objects. Objects allow information flows other than to the return value. Information may flow into an object directly or indirectly referenced by 1) a method argument, 2) a field in the method's class, or 3) the return value. To expand coverage, we extend the heuristic to recognize object references to arrays and Java *String* objects when used as arguments and the return value.

TaintDroid also defines *method profiles*, which are lists of $(from, to)$ pairs indication information flow between variables. The profile may specify method parameters, class variables, and return values. If any of these variables are objects, the profile specifies the object type and allows arbitrary levels of dereferencing by variable name and type. The profile is automatically applied on method termination.

We performed a survey of the JNI methods included in the official Android source code (version 2.1) to determine specific properties. We found 2,844 JNI methods with a Java interface and C or C++ implementation.[3] Of these methods, 913 did not reference objects (as arguments, return value, or method body) and hence are automatically covered by our heuristic. The remaining methods may or may not have information flows that produce false negatives. Future work will provide a more indepth static analysis to identify flows and automatically generate method profiles. Currently, we define method profiles as needed. For example, methods in the IBM *NativeConverter* class require propagation for conversion between character and byte arrays. These methods are frequently used when transmitting strings over network connections.

---

[3] There was a relatively small number of JNI methods that did not either have a Java interface or C/C++ implementation. These unusable methods were excluded from our survey.

5.4.4 IPC Taint Propagation

Taint tags must propagate between applications when they exchange data. The tracking granularity affects performance and memory overhead. TaintDroid uses message-level taint tracking. A message taint tag represents the upper bound of taint markings assigned to variables contained in the message. We use message-level granularity to minimize performance and storage overhead during IPC.

We modified the C++ parcel message object to store one taint tag per parcel and added two interface methods: *updateTaint()* and *getTaint()*. The former method unions its argument tag value with the existing parcel taint tag. The latter method retrieves parcel's current taint tag. We then modified the Java parcel object with shims for all marshall (e.g., *writeInt()*) and unmarshall (e.g., *readInt()*) methods. The shims use our taint library (Section 5.4.6) to acquire and set taint tags on Java variables. We modified the Java interface, because the C++ JNI interface cannot access argument taint tags.

The binder IPC mechanism transfers C++ parcel objects. The IPC transmission passes the byte array maintained by a parcel to the binder kernel module, which copies the memory into the remote process. TaintDroid appends the parcel taint tag to the byte array immediately before transmission to the kernel and sets the taint tag of the parcel object in the remote process upon receipt and requires no kernel modifications.

We chose to implement message-level over variable-level taint propagation, because in a variable-level system, a devious receiver could game the monitoring by unpacking variables in a different way to acquire values without taint propagation. For example, if an IPC parcel message contains a sequence of scalar values, the receiver may unpack a string instead, thereby acquiring values without propagating all the taint tags on scalar values in the sequence. Hence, to prevent applications from removing taint tags in this way, the current implementation protects taint tags at the message-level.

Message-level taint propagation for IPC leads to false positives. Similar to arrays, all data items in a parcel share the same taint tag. For example, Section 5.8 discusses limitations for tracking the IMSI that results from passing as portions the value as configuration parameters in parcels. Future implementations will consider word-level taint tags along with additional consistency checks to ensure accurate propagation for unpacked variables. However, this additional complexity will negatively impact IPC performance.

5.4.5 Secondary Storage Taint Propagation

Taint tags may be lost when data is written to a file. Our design stores one taint tag per file. The taint tag is updated on file write and propagated to data on file read. TaintDroid stores file taint tags in the file system's extended attributes. To do this, we implemented extended attribute support for Android's host file system (YAFFS2) and formatted the removable SDcard with the ext2 file system. As with arrays and IPC, storing one taint tag per file leads to false positives and limits the granularity of taint markings for information databases (see Section 5.5). Alternatively, we could track taint tags at a finer granularity at the expense of added memory and performance overhead.

5.4.6 Taint Interface Library

Taint sources and sinks defined within the virtualized environment must communicate taint tags with the tracking system. We abstract the taint source and sink logic into a single taint interface library. The interface performs two functions: 1) add taint markings to variables; and 2) retrieve taint markings from variables. The library only provides the ability to add and not set or clear taint tags, as such functionality could be used by untrusted Java code to remove taint markings.

Adding taint tags to arrays and strings via internal VM methods is straightforward, as both are stored in data objects. Primitive type variables, on the other hand, are stored on the interpreter's internal stack and disappear after a method is called. Therefore, the taint library uses the method return value as a means of tainting primitive type variables. The developer passes a value or variable into the appropriate add taint method (e.g., *addTaintInt()*) and the returned variable has the same value but additionally has the specified taint tag. Note that the stack storage does not pose complications for taint tag retrieval.

5.5 Privacy Hook Placement

Using TaintDroid for privacy analysis requires identifying privacy sensitive sources and instrumenting taint sources within the operating system. Historically, dynamic taint analysis systems assume taint source and sink placement is trivial. However, complex operating systems such as Android provide applications information in a variety of ways, e.g., direct access, and service interface. Each potential type of privacy sensitive information must be studied carefully to determine the best method of defining the taint source.

Taint sources can only add taint tags to memory for which TaintDroid provides tag storage. Currently, taint source and sink placement is limited to variables in interpreted code, IPC messages, and files. This section discusses how valuable taint sources and sinks can be implemented within these restrictions. We generalize such taint sources based on information characteristics.

Low-bandwidth Sensors: A variety of privacy sensitive information types are acquired through low-bandwidth sensors, e.g., location and accelerometer. Such information often changes frequently and is simultaneously used by multiple applications. Therefore, it is common for a smartphone OS to multiplex access to low-bandwidth sensors using a manager. This sensor manager represents an ideal point for taint source hook placement. For our analysis, we placed hooks in Android's LocationManager and SensorManager applications.

High-bandwidth Sensors: Privacy sensitive information sources such as the microphone and camera are high-bandwidth. Each request from the sensor frequently returns a large amount of data that is only used by one application. Therefore, the smartphone OS may share sensor information via large data buffers, files, or both. When sensor information is shared via files, the file must be tainted with the appropriate tag. Due to flexible APIs, we placed hooks for both data buffer and file tainting for tracking microphone and camera information.

Information Databases: Shared information such as address books and SMS messages are often stored in file-based databases. This organization provides a useful unambiguous taint source similar to hardware sensors. By adding a taint tag to such database files, all information read from the file will be automatically tainted. We used this technique for tracking address book information. Note that while TaintDroid's file-level granularity was appropriate for these valuable information sources, others may exist for which files are too coarse grained. However, we have not yet encountered such sources.

Device Identifiers: Information that uniquely identifies the phone or the user is privacy sensitive. Not all personally identifiable information can be easily tainted. However, the phone contains several easily tainted identifiers: the phone number, SIM card identifiers (IMSI, ICC-ID), and device identifier (IMEI) are all accessed through well-defined APIs. We instrumented the APIs for the phone number, ICC-ID, and IMEI. An IMSI taint source has inherent limitations discussed in Section 5.8.

Table 5.2. Applications grouped by the requested permissions (L: location, C: camera, A: audio, P: phone state). Android Market categories are indicated in parenthesis, showing the diversity of the studied applications.

Applications[*]	#	**Permissions**[†]			
		L	C	A	P
The Weather Channel (News & Weather); Cestos, Solitaire (Game); Movies (Entertainment); Babble (Social); Manga Browser (Comics)	6	x			
Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic Jam, Hearts, Blackjack, (Games); Horoscope (Lifestyle); Yellow Pages (Reference); 3001 Wisdom Quotes Lite, Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment)	14	x			x
Layar (Lifestyle); Knocking (Social); Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports)	6	x	x		x
MySpace (Social); Barcode Scanner, ixMAT (Shopping)	3		x		
Evernote (Productivity)	1	x	x	x	

[*] Listed names correspond to the name displayed on the phone and not necessarily the name listed in the Android Market.

[†] All listed applications also require access to the Internet.

Network Taint Sink: Our privacy analysis identifies when tainted information transmits out the network interface. The VM interpreter-based approach requires the taint sink to be placed within interpreted code. Hence, we instrumented the Java framework libraries at the point the native socket library is invoked.

5.6 Application Study

This section reports on an application study that uses TaintDroid to analyze how 30 popular third-party Android applications use privacy sensitive user data. Existing applications acquire a variety of user data along with permissions to access the Internet. Our study finds that two thirds of these applications expose detailed location data, the phone's unique ID, and the phone number using the combination of the seemingly innocuous access permissions granted at install. This finding was made possible by TaintDroid's ability to monitor runtime *access* of sensitive user data and to precisely relate the monitored accesses with the *data exposure* by applications.

5.6.1 Experimental Setup

An early 2010 survey of the 50 most popular free applications in each category of the Android Market [206] (1,100 applications, in total) revealed that roughly a third of the applications (358 of the 1,100 applications) require Internet permissions along with permissions to access either location, camera, or audio data. From this set, we randomly selected 30 popular applications (an 8.4% sample size), which span twelve categories. Table 5.2 enumerates these applications along with permissions they request at install time. Note that this does not reflect actual access or use of sensitive data.

We studied each of the thirty downloaded applications by starting the application, performing any initialization or registration that was required, and then manually exercising the functionality offered by the application. We recorded system logs including detailed information from TaintDroid: tainted binder messages, tainted file output, and tainted network messages with the remote address. The overall experiment (conducted in May 2010) lasted slightly over 100 minutes, generating 22,594 packets (8.6MB) and 1,130 TCP connections. To verify our results, we also logged the network traffic using tcpdump on the WiFi interface and repeated experiments on multiple Nexus One phones, running the same version of TaintDroid built on Android 2.1. Though the phones used for experiments had a valid SIM card installed, the SIM card was inactivate, forcing all the packets to be transmitted via the WiFi interface. The packet trace was used only to verify the exposure of tainted data flagged by TaintDroid.

In addition to the network trace, we also noted whether applications acquired user consent (either explicit or implicit) for exporting sensitive information. This provides additional context information to identify possible privacy violations. For example, by selecting the “use my location” option in a weather application, the user implicitly consents to disclosing geographic coordinates to the weather server.

5.6.2 Findings

Table 5.3 summarizes our findings. TaintDroid flagged 105 TCP connections as containing tainted privacy sensitive information. We manually labeled each message based on available context, including remote server names and temporally relevant application log messages. We used remote hostnames as an indication of whether data was being sent to a server providing application functionality or to a third party. Frequently, messages contained plaintext that aided categorization, e.g., an HTTP GET request containing geographic coordinates. However, 21 flagged messages contained binary data. Our investigation indicates these messages were generated by the Google Maps for Mobile [211] and FlurryAgent [212] APIs and contained tainted privacy sensitive data. These conclusions are supported by message transmissions immediately after the application received a tainted parcel from the system location manager. We now expand on our findings for

Table 5.3. Potential privacy violations by 20 of the studied applications. Note that three applications had multiple violations, one of which had a violation in all three categories.

Observed Behavior (# of apps)	**Details**
Phone Information to Content Servers (2)	2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server.
Device ID to Content Servers (7)*	2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app's content server.
Location to Advertisement Servers (15)	5 apps sent geo-coordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com) and 4 apps sent location† to data.flurry.com.

* TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

† To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).

each category and reflect on potential privacy violations.

Phone Information: Table 5.2 shows that 20 out of the 30 applications require permissions to read phone state and the Internet. We found that 2 of the 20 applications transmitted to their server (1) the device's phone number, (2) the IMSI which is a unique 15-digit code used to identify an individual user on a GSM network, and (3) the ICC-ID number which is a unique SIM card serial number. We verified messages were flagged correctly by inspecting the plaintext payload.[4] In neither case was the user informed that this information was transmitted off the phone.

This finding demonstrates that Android's coarse-grained access control provides insufficient protection against third-party applications seeking to collect sensitive data. Moreover, we found that one application transmits the phone information *every time* the phone boots. While this application displays a terms of use on first use, the terms of use does not specify collection of this highly sensitive data. Surprisingly, this application transmits the phone data immediately after install, before first use.

Device Unique ID: The device's IMEI was also exposed by applications. The IMEI uniquely identifies a specific mobile phone and is used to prevent a stolen handset from accessing the cellular network. TaintDroid flags indicated that nine applications transmitted the IMEI. Seven out of the nine applications either do not present an End User

---

[4] Because of the limitation of the IMSI taint source as discussed in Section 5.8, we disabled the IMSI taint source for experiments. Nonetheless, TaintDroid's flag of the ICC-ID and the phone number led us to find the IMSI contained in the same payload.

License Agreement (EULA) or do not specify IMEI collection in the EULA. One of the seven applications is a popular social networking application and another is a location-based search application. Furthermore, we found two of the seven applications include the IMEI when transmitting the device's geographic coordinates to their content server, potentially repurposing the IMEI as a client ID.

In comparison, two of the nine applications treat the IMEI with more care, thus we do not classify them as potential privacy violators. One application displays a privacy statement that clearly indicates that the application collects the device ID. The other uses the hash of the IMEI instead of the number itself. We verified this practice by comparing results from two different phones.

Location Data to Advertisement Servers: Half of the studied applications exposed location data to third-party advertisement servers without requiring implicit or explicit user consent. Of the fifteen applications, only two presented a EULA on first run; however neither EULA indicated this practice. Exposure of location information occurred both in plaintext and in binary format. The latter highlights TaintDroid's advantages over simple pattern-based packet scanning. Applications sent location data in plaintext to admob.com, ad.qwapi.com, ads.mobclix.com (11 applications) and in binary format to FlurryAgent (4 applications). The plaintext location exposure to AdMob occurred in the HTTP GET string:

```
...&s=a14a4a93f1e4c68&..&t=062A1CB1D476DE85B717D9195A6722A9
&d%5Bcoord%5D=47.661227890000006%2C−122.31589477&...
```

Investigating the AdMob SDK revealed the `s=` parameter is an identifier unique to an application publisher, and the `coord=` parameter provides the geographic coordinates.

For FlurryAgent, we confirmed location exposure by the following sequence of events. First, a component named "FlurryAgent" registers with the location manager to receive location updates. Then, TaintDroid log messages show the application receiving a tainted parcel from the location manager. Finally, the application reports "sending report to `http://data.flurry.com/aar.do`" after receiving the tainted parcel.

Our experimentation indicates these fifteen applications collect location data and send it to advertisement servers. In some cases, location data was transmitted to advertisement servers even when no advertisement was displayed in the application. However, we note that TaintDroid helped us verify that three of the studied applications (not included in the Table 5.3) only transmitted location data per user's request to pull localized content from their servers. This finding demonstrates the importance of monitoring exercised functionality of an application that reflects how the application *actually* uses or abuses the granted permissions.

Legitimate Flags: Out of 105 connections flagged by TaintDroid, 37 were deemed clearly legitimate use. The flags resulted from four applications and the OS itself while using the Google Maps for Mobile (GMM) API. The TaintDroid logs indicate an HTTP request with the "User-Agent: GMM . . . " header, but a binary payload. Given that GMM functionality includes downloading maps based on geographic coordinates, it is obvious that TaintDroid correctly identified location information in the payload. Our manual inspection of each message along with the network packet trace confirmed that there were no false positives. We note that there is a possibility of false negatives, which is difficult to verify with the lack of the source code of the third-party applications.

Summary: Our study of 30 popular applications shows the effectiveness of the TaintDroid system in accurately tracking applications' use of privacy sensitive data. While monitoring these applications, TaintDroid generated no false positives (with the exception of the IMSI taint source which we disabled for experiments, see Section 5.8). The flags raised by TaintDroid helped to identify potential privacy violations by the tested applications. Half of the studied applications share location data with advertisement servers. Approximately one third of the applications expose the device ID, sometimes with the phone number and the SIM card serial number. The analysis was simplified by the taint tag provided by TaintDroid that precisely describes which privacy relevant data is included in the payload, especially for binary payloads. We also note that there was almost no perceived latency while running experiments with TaintDroid.

5.7 Performance Evaluation

We now study TaintDroid's taint tracking overhead. Experiments were performed on a Google Nexus One running Android OS version 2.1 modified for TaintDroid. Within the interpreted environment, TaintDroid incurs the same performance and memory overhead regardless of the existence of taint markings. Hence, we only need to ensure file access includes appropriate taint tags.

5.7.1 Macrobenchmarks

During the application study, we anecdotally observed limited performance overhead. We hypothesize that this is because: 1) most applications are primarily in a "wait state,"

Table 5.4. Macrobenchmark Results

Benchmark	**Android**	**TaintDroid**
App Load Time	63 ms	65 ms
Address Book (create)	348 ms	367 ms
Address Book (read)	101 ms	119 ms
Phone Call	96 ms	106 ms
Take Picture	1718 ms	2216 ms

and 2) heavyweight operations (e.g., screen updates and webpage rendering) occur in unmonitored native libraries.

To gain further insight into perceived overhead, we devised five macrobenchmarks for common high-level smartphone operations. Each experiment was measured 50 times and observed 95% confidence intervals at least an order of magnitude less than the mean. In each case, we excluded the first run to remove unrelated initialization costs. Experimental results are shown in Table 5.4.

Application Load Time: The application load time measures from when Android's Activity Manager receives a command to start an activity component to the time the activity thread is displayed. This time includes application resolution by the Activity Manager, IPC, and graphical display. TaintDroid adds only 3% overhead, as the operation is dominated by native graphics libraries.

Address Book: We built a custom application to create, read, and delete entries for the phone's address book, exercising both file read and write. Create used three SQL transactions while read used two SQL transactions. The subsequent delete operation was lazy, returning in 0 ms, and hence was excluded from our results. TaintDroid adds approximately 5.5% and 18% overhead for address book entry creates and reads, respectively. The additional overhead for reads can be attributed to file taint propagation. The data is not tainted before create, hence no file propagation is needed. Note that the user experiences less than 20 ms overhead when creating or viewing a contact.

Phone Call: The phone call benchmark measured the time from pressing "dial" to the point at which the audio hardware was reconfigured to "in call" mode. TaintDroid only adds 10 ms per phone call setup (~10% overhead), which is significantly less than call setup in the network, which takes on the order of seconds.

Take Picture: The picture benchmark measures from the time the user presses the "take picture" button until the preview display is re-enabled. This measurement includes

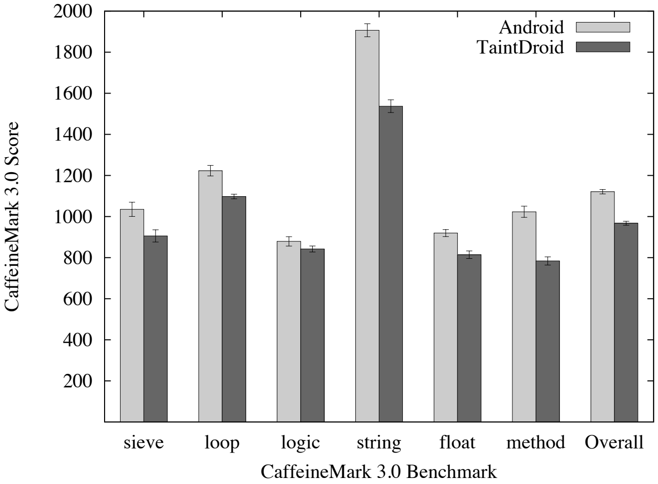

Figure 5.5. Microbenchmark of Java overhead. Error bars indicate 95% confidence intervals.

the time to capture a picture from the camera and save the file to the SDcard. TaintDroid adds 498 ms to the 1718 ms needed by Android to take a picture (an overhead of 29%). A portion of this overhead can be attributed to additional file operations required for taint propagation (one *getxattr*/*setxattr* pair per written data buffer). Note that some of this overhead can be reduced by eliminating redundant taint propagation. That is, only the taint tag for the first data buffer written to file needs to be propagated. For example, the current taint tag could be associated with the file descriptor.

5.7.2 Java Microbenchmark

Figure 5.5 shows the execution time results of a Java microbenchmark. We used an Android port of the standard CaffeineMark 3.0 [213]. CaffeineMark uses an internal scoring metric only useful for relative comparisons.

The results are consistent with implementation-specific expectations. The overhead incurred by TaintDroid is smallest for the benchmarks dominated by arithmetic and logic operations. The taint propagation for these operations is simple, consisting of an additional copy of spatially local memory. The string benchmark, on the other hand, experiences the greatest overhead. This is most likely due to the additional memory comparisons that occur when the JNI propagation heuristic checks for string objects in method prototypes.

The "overall" results indicate cumulative score across individual benchmarks. CaffeineMark documentation states that scores roughly correspond to the number of Java instructions executed per second. Here, the unmodified Android system had an aver-

Table 5.5. IPC Throughput Test (10,000 msgs).

	Android	**TaintDroid**
Time (s)	8.58	10.89
Memory (client)	21.06MB	21.88MB
Memory (service)	18.92MB	19.48MB

age score of 1121, and TaintDroid measured 967. TaintDroid has a 14% overhead with respect to the unmodified system.

We also measured memory consumption during the CaffeineMark benchmark. The benchmark consumed 21.28 MB on the unmodified system and 22.21 MB while running on TaintDroid, indicating a 4.4% memory overhead. Note that much of an Android process's memory is used by the zygote runtime environment. These native library memory pages are shared between applications to reduce the overall system memory footprint and require taint tracking. Given that TaintDroid stores 32 taint markings (4 bytes) for each 32-bit variable in the interpreted environment (regardless of taint state), this overhead is expected.

5.7.3 IPC Microbenchmark

The IPC benchmark considers overhead due to the parcel modifications. For this experiment, we developed client and service applications that perform binder transactions as fast as possible. The service manipulates account objects (a username string and a balance integer) and provides two interfaces: *setAccount()* and *getAccount()*. The experiment measures the time for the client to invoke each interface pair 10,000 times.

Table 5.5 summarizes the results of the IPC benchmark. TaintDroid was 27% slower than Android. TaintDroid only adds four bytes to each IPC object, therefore overhead due to data size is unlikely. The more likely cause of the overhead is the continual copying of taint tags as values are marshalled into and out of the parcel byte buffer. Finally, TaintDroid used 3.5% more memory than Android, which is comparable to the consumption observed during the CaffeineMark benchmarks.

5.8 Discussion

Approach Limitations: TaintDroid only tracks data flows (i.e., explicit flows) and does not track control flows (i.e., implicit flows) to minimize performance overhead. Section 5.6 shows that TaintDroid can track applications' expected data exposure and also reveal suspicious actions. However, applications that are truly malicious can game our system and exfiltrate privacy sensitive information through control flows. Fully tracking control flow requires static analysis [144, 140], which is not applicable to analyzing third-party applications whose source code is unavailable. Direct control flows can be tracked dynamically if a taint scope can be determined [165]; however, DEX does not maintain branch structures that TaintDroid can leverage. On-demand static analysis to determine method control flow graphs (CFGs) provides this context [164]; however, TaintDroid does not currently perform such analysis in order to avoid false positives and significant performance overhead. Our data flow taint propagation logic is consistent with existing, well known, taint tracking systems [155, 148]. Finally, once information leaves the phone, it may return in a network reply. TaintDroid cannot track such information.

Implementation Limitations: Android uses the Apache Harmony [208] implementation of Java with a few custom modifications. This implementation includes support for the *PlatformAddress* class, which contains a native address and is used by *DirectBuffer* objects. The file and network IO APIs include write and read "direct" variants that consume the native address from a *DirectBuffer*. TaintDroid does not currently track taint tags on *DirectBuffer* objects, because the data is stored in opaque native data structures. Currently, TaintDroid logs when a read or write "direct" variant is used, which anecdotally occurred with minimal frequency. Similar implementation limitations exist with the *sun.misc.Unsafe* class, which also operates on native addresses.

Taint Source Limitations: While TaintDroid is very effective for tracking sensitive information, it causes significant false positives when the tracked information contains configuration identifiers. For example, the IMSI numeric string consists of a Mobile Country Code (MCC), Mobile Network Code (MNC), and Mobile Station Identifier Number (MSIN), which are all tainted together.[5] Android uses the MCC and MNC extensively as configuration parameters when communicating other data. This causes all information in a parcel to become tainted, eventually resulting in an explosion of tainted information. Thus, for taint sources that contain configuration parameters, tainting individual variables within parcels is more appropriate. However, as our analysis results in Section 5.6 show, message-level taint tracking is effective for the majority of our taint sources.

---

[5] Regardless of the string separation, the MCC and MNC are identifiers that warrant taint sources.

5.9 Summary

While some mobile phone operating systems allow users to control applications' access to sensitive information, such as location sensors, camera images, and contact lists, users lack visibility into how applications use their private data. To address this, we presented TaintDroid, an efficient, system-wide information flow tracking tool that can simultaneously track multiple sources of sensitive data. A key design goal of TaintDroid is efficiency, and TaintDroid achieves this by integrating four granularities of taint propagation (variable-level, message-level, method-level, and file-level) to achieve a 14% performance overhead on a CPU-bound microbenchmark.

We also used our TaintDroid implementation to study the behavior of 30 popular third-party applications, chosen at random from the Android Marketplace. Our study revealed that two-thirds of the applications in our study exhibit suspicious handling of sensitive data, and that 15 of the 30 applications reported users' locations to remote advertising servers. Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with monitoring tools such as TaintDroid.

Chapter 6

Static Analysis of Smartphone Application Source Code

The previous chapters discussed what applications can do based on permissions, and how applications use permissions to access privacy sensitive information based on runtime analysis. We found in Chapter 4 that static configuration-level analysis alone does not always provide sufficient context to distinguish benign and malicious functionality. Chapter 5 demonstrated that dynamic taint analysis can practically increase operational context; however it is limited to: *a*) malicious and dangerous behavior based on misuse of information; and *b*) functionality exercised during the analysis process. This chapter similarly seeks to derive greater operational context from applications, but does so based on application source code. This approach has two advantages over the dynamic taint analysis presented in Chapter 5. First, we can look for more than just information misuse. As described in the following sections, we also look for dangerous control flows within API use. We also use control flows to identify vulnerabilities in applications. Second, we are not limited to exercised functionality. The analysis statically considers the application's entire source code, thereby addressing the question, *what can an application do with permissions based on implemented functionality?*.

6.1 A Study of Android Application Security

The fluidity of application markets presents enormous security challenges. Rapidly developed and deployed applications [214], coarse permission systems [5], privacy-invading behaviors [6, 32, 215], malware [216], and limited security models [118, 117, 217] have led to exploitable phones and applications. Although users seemingly desire it, application markets are not in a position to provide security in more than a superficial way [2]. The lack of a common definition for security and the volume of applications render a market-based analysis a Gordian knot.

Understand that malicious, questionable, and vulnerable applications will always find their way through attempts to certify applications before market distribution. Embracing this fact, third-party security services such as the App Genome Project [200] and WhatApp [201] provide security and privacy reviews of applications across markets. Undercutting academic and industry efforts is a pervasive lack of transparency. Ongoing efforts have looked at install time artifacts (e.g., permissions [5]), or run-time behaviors (e.g., taint-tracking [6]) to grossly estimate security. More recent efforts have begun to evaluate code by analyzing application binaries for specific behaviors [32]. However, these attempts have failed to broadly address a core question in smartphone security: *do consumer applications contain vulnerabilities or malicious code?*

In this chapter, we broadly characterize the security of applications in the Android Market. In contrast to past studies with narrower foci, e.g., [6, 32], we consider a breadth of security concerns including both dangerous functionality and vulnerabilities, and apply a wide range of analysis techniques. We briefly introduce the Dalvik decompilation (`ded`) process that recovers source code from installation packages. We subject current popular Android applications to a battery of general and platform-specific source code analyses targeted at identifying poor security design, vulnerabilities, and malicious behaviors. We document the results and provide an initial characterization of the state of security in Android applications. This chapter makes the following contributions:

- We create an extensive test suite for analyzing the security of Android applications. These tests are formulated as control flow, data flow, structural, and semantic source code analysis queries.
- We execute the security analysis on 21 million lines of source code from the top 1,100 free applications in the Android application market. Results are manually verified by investigating the code identified by static analysis.
- We document the results of the analysis and identify the root cause and potential severity of discovered vulnerabilities. We posit on the current state of security in Android applications and consider areas of future analysis.

Our popularity-focused security analysis provides important insight into the most frequently used applications. Our findings inform the following broad observations.

1. Similar to past studies, we found wide misuse of privacy sensitive information—particularly phone identifiers and geographic location. Phone identifiers, e.g., IMEI, IMSI, and ICC-ID, were used for everything from "cookie-esque" tracking to accounts numbers.
2. We found no evidence of telephony misuse, background recording of audio or video, abusive connections, or harvesting lists of installed applications.
3. Ad and analytic network libraries are integrated with 51% of the applications studied, with Ad Mob (appearing in 29.09% of apps) and Google Ads (appearing in 18.72% of apps) dominating. Many applications include more than one of these ad networks.
4. Many developers fail to securely use Android APIs. These failures generally fall into the classification of insufficient protection of privacy sensitive information. However, we found no exploitable vulnerabilities that can lead malicious control of the phone.

This study is an initial but not final word on Android application security. The study described throughout is deep but not exhaustive. Thus, one should be circumspect about any interpretation of the following results as a definitive statement about how secure applications are today. Rather, we believe these results are indicative of the current state, but there remain many aspects of the applications that warrant deeper analysis.

The remainder of this chapter proceeds as follows. Section 6.2 provides an overview of the `ded` decompiler, describing how an Android application's source code can be extracted entirely from its installation package. Section 6.3 describes our analysis methodology and an overview of the areas analyzed in our study. Section 6.4 enumerates our security queries in detail. Section 6.5 presents the results of our analysis including observations made through manual inspection of source code. Section 6.7 summarizes our findings.

6.2 Overview of `ded`

`ded` recovers an application's source code from the installation image. For brevity, here we only highlight the challenges and structure of `ded`. Interested readers are referred to the related paper describing `ded` for a detailed treatment of its design and validation [7].

Android applications are written in Java, but run on the Dalvik virtual machine. The DVM was designed for resource constrained mobile phones. For this reason, the DVM bytecode and run-time environment differ substantially from that for existing JVMs.

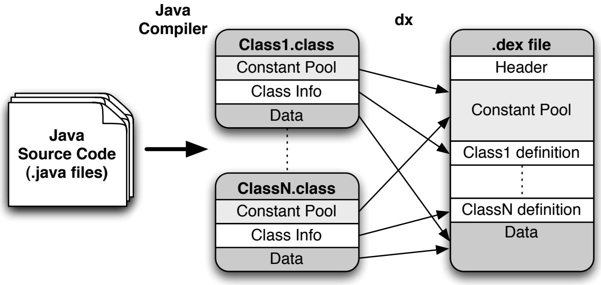

Figure 6.1. Compilation process for DVM applications.

For example, DVM has a different instruction set and constant pool structure, is based on a register architecture (rather than a stack architecture), and has different typing structures. Moreover, in contrast to Java's per-class file structure, a Dalvik application consists of a single `.dex` file containing all classes for the application.

Android applications are compiled using both a Java compiler and the `dex` translator, as shown in Figure 6.1. Here, the Java compiler operates normally to produce a collection of `.class` files. The Dalvik `dx` compiler then consumes the classes, recompiles them to Dalvik bytecode, and writes the resulting application into a single `.dex` file. The `.dex` file is packaged with the application manifest into the installation image `.apk` file.

`ded` extracts an application's source solely from its `.apk` image. This occurs in three stages: *a*) retargeting, *b*) optimization, and *c*) decompilation. `ded` initially retargets Dalvik `.dex` files to Java `.class` files by translating instructions and target/offset references, inferring types, and performing complex class, method, and code reorganization. However, retargeting process yields complex, unoptimized Java bytecode. When decompiled into Java, this unoptimized bytecode becomes a jumbled mess that when run is semantically identical to the original program, but is nearly impossible to visually inspect or analyze using automated tools. For this reason, we use Soot framework [218] to first optimize the ded bytecode output and thereafter decompile it back into the original source.

`ded` has been extensively validated [7]. An initial battery of tests recovered the source code for small, medium and large open source applications and found no errors in recovery. In most cases the recovered code was virtually indistinguishable from the original source (modulo comments and method and variable names, which are not included in the bytecode). A second study of 12 million lines of code for 143 thousand classes in 1,100 applications was performed in late spring of 2010. This study identified 543 individual errors falling into three classes: *a*) unresolved class references caused by incomplete information about system libraries, *b*) type violations present in the Dalvik compiled code (which appears to be bug in `dex`) and *c*) very infrequent pathological boundary cases in which `ded` produces illegal bytecode. Note that all errors are manifest during/after decompilation, and thus we omit the source code of any class containing these errors from study. The number of errors we encountered during this study was vanishingly small, and thus they had no meaningful effect on the study or its results.

The Soot framework uses sophisticated type inference and code analysis techniques [219, 220, 221] to accurately extract program semantics and recover the original code. Soot is centrally an optimization tool with the ability to recover source code in most cases, but does not process certain legal program idioms (bytecode structures) generated by ded. We encountered and refactored many of these idioms while developing ded, but were unable to resolve all of them. In particular, two central problems we have encountered involve interactions between synchronized blocks and exception handling, and complex control flows caused by break statements. The difference between the success rates of retargeted and recovered rates is almost entirely due to Soot's inability to extract source code from these otherwise legal idioms. Closed source decompilers such Jad [222], JD [223] and Fernflower [224] may be more effective at recovering source code, but are either incompatible with the Java bytecode version produced by `ded` or do not provide stand-alone tools. The limitations of Soot places a upper limit on our code recovery success rate, and we will consider other tools in future work.

6.3 Evaluating Android Security

Our Android application study consisted of a broad range of tests focused on three kinds of analysis: *a*) exploring issues uncovered in previous studies and malware advisories, *b*) searching for general coding security failures, and *c*) exploring misuse/security failures in the use of Android framework. The following discusses the process of identifying and encoding the tests.

6.3.1 Analysis Specification

We used four approaches to evaluate recovered source code: *control flow analysis*, *data flow analysis*, *structural analysis*, and *semantic analysis* to perform the security tests, as follows. Unless otherwise specified, all tests used the Fortify SCA [225] static analysis suite of tools.

Control flow analysis: Control flow analysis imposes constraints on the sequences of

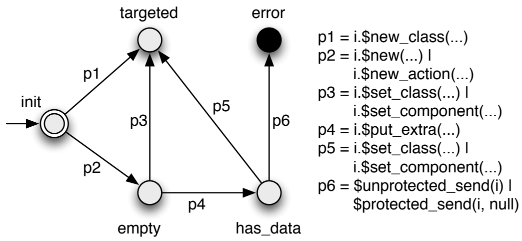

Figure 6.2. Example control flow specification

actions executed by an input program P, classifying some of them as errors. Essentially, a control flow rule is an automaton A whose input words are sequences of actions of P—i.e., the rule *monitors* executions of P. An erroneous action sequence is one that drives A into a predefined *error state*. To statically detect violations specified by A, the program analysis traces each control flow path in the tool's model of P, synchronously "executing" A on the actions executed along this path. Since not all control flow paths in the model are feasible in concrete executions of P, false positives are possible. False negatives are also possible in principle, though uncommon in practice. Figure 6.2 shows an example automaton for sending intents. Here, the error state is reached if the intent contains data and is sent unprotected without specifying the target component, which can result in unintended information leakage.

Data flow analysis: Data flow analysis permits the declarative specification of problematic data flows in the input program. For example, an Android phone contains several pieces of private information that should never leave the phone: the user's phone number, IMEI (device ID), IMSI (subscriber ID), and ICC-ID (SIM card serial number). In our study, we wanted to check that this information is not leaked to the network. While this property can in principle be coded using automata, data flow specification allows for a much easier encoding. The specification declaratively labels program statements matching certain syntactic patterns as *data flow sources* and *sinks*. Data flows between the sources and sinks represent violations.

Structural analysis: Structural analysis allows for declarative pattern matching on the abstract syntax of the input source code. Structural analysis specifications are not concerned with program executions or data flow, therefore, analysis is local and straightforward. For example, in our study, we wanted to specify a bug pattern where an Android application mines the device ID of the phone on which it runs. This pattern was defined using a structural rule that stated that the input program called a method *getDeviceId()* whose enclosing class was *android.telephony.TelephonyManager*.

Semantic analysis: Semantic analysis allows the specification of a limited set of constraints on the values used by the input program. For example, a property of interest in our study was that an Android application does not send SMS messages to hard-coded targets. To express this property, we defined a pattern matching calls to Android messaging methods such as *sendTextMessage()* and *sendDataMessage()*. Semantic specifications permit us to directly specify that the first parameter in these calls (the phone number) is not a constant. The analyzer detects violations to this property using constant propagation techniques well known in program analysis literature.

6.3.2 Analysis Overview

Our analysis covers both dangerous functionality and vulnerabilities. Selecting the properties for study was a significant challenge. The following enumerates our areas of investigation. Section 6.4 discusses the analysis specifications in more detail.

Misuse of Phone Identifiers (Section 6.5.1.1): Previous studies [6, 32] identified phone identifiers leaking to remote network servers. We seek to identify not only the existence of data flows, but to understand why they occur.

Exposure of Physical Location (Section 6.5.1.2): Previous studies [6] identified location exposure to advertisement servers. Many applications provide valuable location-aware utility, which may be desired by the user. By manually inspecting code, we seek to identify the portion of the application responsible for the exposure.

Abuse of Telephony Services (Section 6.5.2.1): Smartphone malware has sent SMS messages to premium-rate numbers. We study the use of hard-coded phone numbers to identify SMS and voice call abuse.

Eavesdropping on Audio/Video (Section 6.5.2.2): Audio and video eavesdropping is a commonly discussed smartphone threat [226]. We examine cases where applications record audio or video without control flows to UI code.

Botnet Characteristics (Sockets) (Section 6.5.2.3): PC botnet clients historically use non-HTTP ports and protocols for command and control. Most applications use HTTP client wrappers for network connections. Therefore, we examine use of the Socket API for suspicious behavior.

Harvesting Installed Applications (Section 6.5.2.4): The list of installed applications is a valuable demographic for marketing. We survey the use of APIs capable of retrieving this list to identify harvesting of installed applications.

Use of Advertisement Libraries (Section 6.5.3.1): Previous studies [6, 32] identified information exposure to ad and analytics networks. We survey inclusion of ad and analytics libraries and the information they access.

Dangerous Developer Libraries (Section 6.5.3.2): During our manual source code inspection, we observed dangerous functionality replicated between applications. We report on this replication and the implications.

Android-specific Vulnerabilities (Section 6.5.4): We search for non-secure coding practices [41, 227], including: writing sensitive information to logs, unprotected broadcasts of information, IPC null checks, injection attacks on intent actions, and delegation.

Misuse of Passwords (Section 6.5.5.1): We look for vulnerabilities in password management, including: hard-coded passwords, null or empty passwords, storage of plaintext passwords, and leakage of passwords to external storage.

Misuse of Cryptography (Section 6.5.5.2): We look for cryptography misuse, including: insufficient key size, deprecated algorithms (e.g., DES, MD5), inappropriate padding, and weak pseudo-random number generators.

Injection Vulnerabilities (Section 6.5.5.3): We look for injection vulnerabilities, including: file paths, databases, and command execution.

6.4 Analysis Query Definitions

We now describe in detail the source code analysis specifications used to identify dangerous functionality and vulnerabilities. We define control flow analysis as FSMs, and data flow analysis as articulations of sources and sinks. The structural and semantic are presented in the Fortify SCA syntax used for the analysis. We begin with analysis definitions for dangerous functionality.

6.4.1 Dangerous Functionality

Our analysis considers several types of dangerous functionality. First, we discuss exfiltration of information using data flow analysis. Specifically, we consider location and phone identifiers as data flow sources. Second, we discuss the use of semantic analysis to identify misuse of telephony services by looking for static values passed to corresponding APIs. Third, we describe how to identify recording of audio and video in the background using both structural and control flow analysis. Fourth, we discuss how the *Socket* API can be used as an indicator for identifying botnet-like activity. Here, we use structural analysis as an more-intelligent *grep* utility. The analysis results in Section 6.5 demonstrate how even simple queries can shed enormous insight. Finally, we discuss specifications to help identify harvesting of the list of installed applications.

Note that all of the specifications are heuristics in and of themselves. They represent idioms that indicate potential dangerous functionality. The goal of study is to *practically* understand functionality by any means necessary. Hence, we use program analysis to minimize manual inspection effort. As will be shown in the results for several rules, a small number of false positives is worth the manual effort required to identify the non-existence of a dangerous behavior.

6.4.1.1 Exfiltration of Information

We use data flow analysis to identify exfiltration of phone identifiers and location. Our specification uses the following data flow sources in sinks. Note that there are several APIs through which information may be transmitted to the network. As discussed below, subtleties of these APIs require creative definitions.

Additionally, note that the Fortify SCA data flow specification syntax allows us to apply source and sink targets to all classes that implement, override, and extend the specified class. This syntax significantly simplifies definition specification.

Data Flow Sources: We define a data flow source for phone identifier APIs of interest: *getLine1Number()* (phone number); *getDeviceID()* (IMEI); *getSubscriberId()* (IMSI); and *getSimSerialNumber()* (ICC-ID). All of these APIs are defined within the *TelephonyManager* class in the *android.telephony* namespace.

Location information can be accessed directly, or obtained via a callback method registered with the phone's location manager. Instead of using these APIs as data sources, we observe that location information is always obtained as a *Location* object defined in the *android.location* namespace. The latitude and longitude values are obtained by invoking the *getLatitude()* and *getLongitude()* access methods. Therefore, we use these methods as data flow sources. While this choice may result in additional false positives, it simplifies source specification, and possibly more importantly, shortens the data flow path that must be tracked. This latter advantage is important, because it lessens the impact of Android IPC and non-recoverable source code.

Data Flow Sinks: The network is our primary data sink; however, there are multiple APIs through which information can leave an application. Specifically, Android provides HTTP-specific APIs to simplify communication with Web servers. We consider both the *URLConnection* class in the *java.net* namespace (and extended classes) as well as the *HttpClient* class in *org.apache.http.client* namespace.

Our data flow sinks consider HTTP GET and POST parameters, request properties, and data written to the output stream. Specification of parameters and properties to APIs is straightforward. First, HTTP GET parameters are commonly encoded directly in a URL. Therefore, we define a data flow sink at the constructor and *set()* method of the *URL* class in the *java.net* namespace. Second, the *HttpClient* API is commonly used to specify HTTP POST parameters. We define a data flow sink at the *setEntity()* method of the *HttpEntityEnclosingRequestBase* class of the *org.apache.http.client.methods* namespace. However, to ensure the data flow is properly tracked, defined several data flow pass-through rules for the constructors the *HttpEntity* and *AbstractHttpEntity* in the *org.apache.http.entity* namespace, and the *BasicNameValuePair* class in the *org.apache.http.message* namespace. HTTP POST parameters are passed to these classes, and we must track information through them to *setEntity()*. Since these classes are within the Android library (and not included in the source code analysis), data flow pass-through specifications are required. Finally, we define a data flow sink at the *setRequestProperty()* of the *URLConnection* class in the *java.net* namespace.

In addition to these APIs, sensitive information may be written to an output stream. However, we must distinguish between a file output stream and a network output stream. To distinguish between these two IO interfaces, we define a data flow source at the *getOutputStream()* method of the *URLConnection* in the *java.net* namespace. This source adds the flat `NETOUT` to the returned *OutputStream* object. Next, leveraging the implements/overrides/extends syntactic feature in SCA, we define a pass-through rule for all *OutputStream* and *Writer* classes in the *java.io* namespace. This effectively propagates the `NETOUT` flag to all corresponding output child classes. Finally, we define a data flow sink at the *write()* method of the *OutputStream* and *Writer* classes (again utilizing implements/overrides/extends), to identify the combination of the `NETOUT` flag *and* one of the data flow sources for location and phone identifiers, as discussed above. Figure 6.3 depicts this approach.

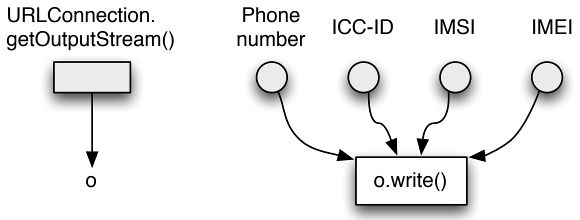

Figure 6.3. Data flow analysis to distinguish file and network IO.

Identification of API Use: The data flow analysis may have false negatives resulting from Android IPC and code recovery. To gain a better understanding of the information that applications access, we also define simple queries for API use. For phone identifiers, our rules identify when the above defined methods are called. For location, we consider both the *getLastKnownLocation()* and *requestLocationUpdates()* methods of the *LocationManager* class in the *android.location* namespace. Additionally, we query for instances of the *onLocationChanged()* callback method defined within a *LocationListener* class in the *android.location* namespace.

6.4.1.2 Misuse of Telephony Services

We use semantic analysis to identify misuse of telephony service APIs. Specifically, we are interested in the destination phone number used for SMS messages and placing voice calls. Android defines significantly different APIs for SMS and voice calls. Use cases are also different. For example, calling statically defined customer service numbers from an application is sometimes desirable, whereas sending SMS messages are generally not used for customer service (as email provides a more suitable mechanism for asynchronous help).

SMS Services: We looked for hard-coded phone numbers in the SMS API using semantic analysis. Our specification identifies constant values passed to the destination phone number in the *sendTextMessage()*, *sendDataMessage()*, and *sendMultiPartTextMessage()* methods of the *SmsManager* class in the *android.telephony* namespace. Note that this specification is limited by Fortify SCA's analysis. The semantic analysis is limited to constant values, which are semantically different than hard-coded values. While the definition identifies hard-coded values passed directly to the API, it does not identify hard-coded values assigned to variables, unless the Java variable is made `final`.

Voice Call Services: Android does not use a simple method for making voice calls.

```
FunctionCall c: c.function.name == "read" and
c.function.enclosingClass.name == "android.media.AudioRecord"
and not c.enclosingClass reachedBy
[ Class a: a.supers contains
[ Class super: super.name == "android.app.Activity"]]
```

Figure 6.4. Structural analysis definition to identify background audio recording.

Instead, an intent with either the `CALL` or `DIAL` framework defined action strings is used to start an activity. This intent specifies a phone number as its data URI field. If the `CALL` action string is used, the application must have the `CALL_PHONE` permission, and the voice call is initiated immediately. If the `DIAL` action string is used, Android starts the phone dialer with the specified number entered, but the user must select the "call" button to initiate the voice call. No permission is required to use the `DIAL` action string.

Lacking an obvious method argument for semantic analysis, we consider the *parse()* method of the *Uri* class in the *java.net* namespace. *Uri* objects are used for various purposes; however, Android uses the "`tel:`" prefix to identify telephone numbers. Therefore, our semantic analysis queries for constant strings beginning with "`tel:`" (note that again, this is an approximation of the desired analysis for hard-coded values). We also define a query that includes "`900`" in the semantic analysis to identify premium-rate numbers.

6.4.1.3 Background Audio and Video Recording

Android provides three APIs for recording audio and visual information. The *AudioRecord* class in the *android.media* namespace defines the *read()* method to access microphone input. The *Camera* class in the *android.hardware* namespace defines the *takePicture()* method to capture still images. Finally, the *MediaRecorder* class in the *android.media* namespace is used to both record audio and video.

Audio: Simply recording audio is not suspicious in and of itself. As an approximation of recording audio without the user's knowledge, we look for use of the *AudioRecord* API in code not accessible to an Android activity component. Figure 6.4 provides this structural analysis definition. Here, we use the special `reachedBy` directive that considers the call graph leading to the execution of *read()*.

A nearly identical specification is used for the *start()* method of *MediaRecord*. This rule also identifies video recording; however, as we discuss below, there are alternative methods of identifying background recording of video.

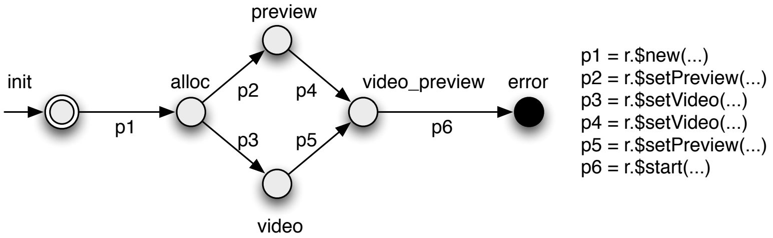

Figure 6.5. FSM for control flow analysis to identify background video recording.

Still Images: Still images are obtained using the *takePicture()* method of the *Camera* class. The API requires that *startPreview()* is called before *takePicture()*. In order to call *takePicture()*, *setPreviewDisplay()* must be called to specify a user interface layout area to display a preview of the still image. Therefore, still images cannot be taken without the user's knowledge.

Video: The *MediaRecorder* class contains methods to set audio and video sources. Our specification ensures that if *setVideoSource()* is called, then so is *setPreviewDisplay()*, which will inform the user that the camera is being accessed. The FSM for our control flow specification is shown in Figure 6.5.

6.4.1.4 Socket API Use

We hypothesize that most network-based Android applications are HTTP clients. Android includes the *URLConnection* and *HttpClient* APIs to abstract network communication to Web servers. Therefore, we expect most applications will choose to use these APIs over direct access with *Socket* objects. To identify the use sockets, we use simple structural analysis that queries invocation of the *connect()* method and the *InetAddress* and *String* constructor variants of the *Socket* class in the *java.net* namespace.

Note that this definition is not looking for dangerous functionality directly. In fact, we expect some applications will use sockets directly, e.g., for streaming audio. Rather, we hope to characterize how many applications use sockets directly in order to evaluate our hypothesis and determine its usefulness as a security analysis heuristic.

6.4.1.5 Harvesting Installed Applications

Android provides the *PackageManager* API (in the *android.content.pm* namespace) to abstract access to information pertaining to installed applications, their components, and permissions. Using this API, an application harvest the list of installed applications.

There are two general approaches to acquiring the list of installed applications. First, one of the *getInstalledApplications()*, *getInstalledPackages()*, or *getPreferredPackages()* can be called. Second, generic queries can be made to one of the *queryIntentActivities()*, *queryIntentServices()*, *queryBroadcastReceivers()*, or *queryContentProviders()* methods. We use a simple structural analysis definition for each of these two cases.

6.4.2 Vulnerabilities

Android applications are written Java. There are many vulnerability definitions for Java applications; however, the definitions are frequently designed to detect vulnerabilities in server applications. While our study includes analysis of standard Java vulnerabilities, we seek to define Android-specific vulnerability specifications. The remainder of this section discusses these specifications. First, we use data flow analysis to identify location and phone identifiers written to Android's insecure logging interface. Second, we use control flow analysis to identify unsafe intent broadcasts that may expose sensitive information. Third, we use control flow analysis to identify insufficiently protected dynamically created broadcast receiver components. Fourth, we use data flow analysis to identify injection attacks on intent messages. Fifth, we use control flow analysis to identify delegation vulnerabilities when using pending intents. Finally, we use control flow analysis to look specifically for failures to check for null values before dereferencing objects obtained via Android's IPC API.

6.4.2.1 Leaking Information to Insecure Locations

We look for two types of information leaks: leaks to log files, and leaks to IPC. We begin with log files.

Leaks to Log files: Java has several APIs for simplified application logging. These log files are often stored within a directory only accessible to the application writing to the log. However, Android includes the *logcat* API for centralized logging by all applications. Any application with the `READ_LOGS` permission can access the logs. For the data flow analysis, we use the same sources for location and phone identifiers indicated in Section 6.4.1.1 to identify exfiltration. We define a data flow sink at all methods of the *Log* class in the *android.util* namespace.

Leaks to IPC: Sending sensitive information over IPC is only a vulnerability if it can be accessed by an unintended application. Therefore, instead of using data flow analysis

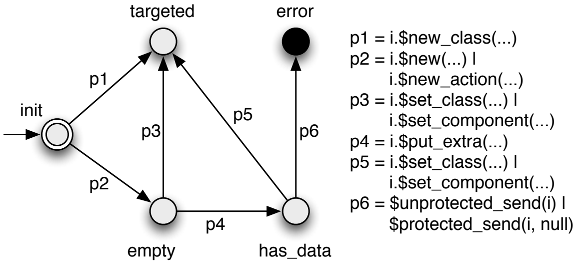

Figure 6.6. FSM for control flow analysis to identify leaks to IPC.

to identify leaks to IPC, we use control flow analysis to determine if an intent message is broadcast to an action string without specifying either a permission to protect the intent, or the target application and component name. If this condition is not met, a malicious application can eavesdrop to potentially access sensitive information. The FSM for this control flow analysis is shown in Figure 6.6.

In the figure, paths lead to the "targeted" state if the application and component is explicitly provided. Note that there are several ways in which this can occur. Next, we only want to identify intent messages that contain "extras" information, therefore, the FSM must transition to `has_data` before reaching the `error` state. We assume intent messages without extras do not contain sensitive information. Finally, eavesdropping can only occurs if the intent is broadcast without a permission protecting its access. Note that the developer may pass `null` to the API accepting a permission.

6.4.2.2 Unprotected Broadcast Receivers

Activity and service components occasionally dynamically register broadcast receivers to receive intent messages only during a specific time interval (e.g., while the activity is in focus). By default, if a component defines an intent filter, it is public, and any application can construct an intent message that can be sent directly to it. In contrast, if there is no intent filter, the component is private, and can only be accessed by components in the same application. Developers can prevent forging attacks by protecting public components with a permission. However, there are two APIs to dynamically register a broadcast receiver, and one does not include a permission to protect the component.

Figure 6.7 shows the FSM for control flow analysis to identify unprotected dynamic registration of a broadcast receiver that has an intent filter. Here, the FSM tracks an intent filter object `if`. The intent filter will only make the broadcast receiver public if it contains at least one action string. Finally, the FSM considers the case where the developer passes `null` as a permission.

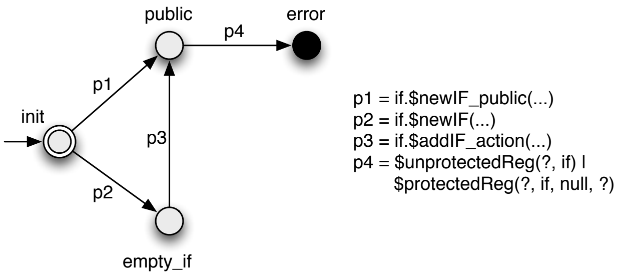
public
error
p4
p1
init
p3
p2
empty_if
p1 = if.$newIF_public(...)
p2 = if.$newIF(...)
p3 = if.$addIF_action(...)
p4 = $unprotectedReg(?, if) |
$protectedReg(?, if, null, ?)

Figure 6.7. FSM for control flow analysis to identify unprotected broadcast receivers.

6.4.2.3 Intent Injection Attacks

Android's intent messages perform actions and therefore subject to injection attacks from the network and IPC input from other applications. Specifically, we are interested in untrusted input that is used in an address field of an intent message, which includes both the action string and the destination application and component fields.

Network Data Flow Sources: To identify untrusted network input, we assign the data flow source flag `NETIN` at the return of the *getInputStream()* method of the *URLConnection* class in the *java.net* namespace, as well as the *getEntity()* method of the *HttpResponse* class in the *org.apache.http* namespace. To account for logic within Android libraries, we define pass-through specifications to propagate `NETIN` through *InputStream* and *Reader* class constructors (*java.io* namespace), and to the return of the *read()* methods of these classes. Note that we again leverage the implements/overrides/extends feature for method specification. Additionally, to propagate `NETIN` for input from *HttpResponse*, we define a pass-through specification for the *toString()*, *toByteArray()*, and *getContentCharSet()* methods of the *EntityUtils* class in the *org.apache.http.util* namespace.

IPC Data Flow Sources: To identify untrusted IPC input, we assign the data flow source flag `IPCIN` to the return of all methods with prefix "*get*" in the *Intent* class of the *android.content* namespace. The prefix matching is achieved using wildcard symbols.

Intent Address Data Flow Sink: We define the data flow sink for `NETIN` and `IPCIN` flags at the inputs to the constructor, *setAction()*, *setClassName()*, and *setComponent()* methods of the *Intent* class in the *android.content* namespace. Note that we define an additional pass-through specification for the constructor of the *ComponentName* class of the *android.content* namespace. This class converts text strings specifying the application and component names into a *ComponentName* object passed to *setComponent()*.

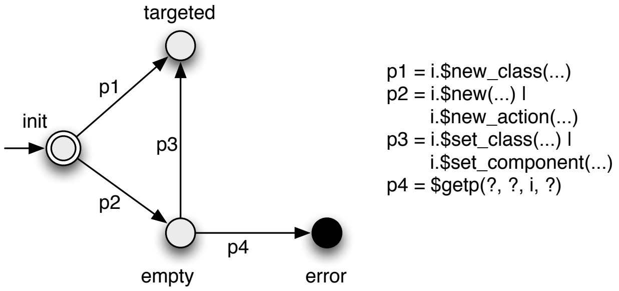

Figure 6.8. FSM for control flow analysis to identify unsafe pending intents.

6.4.2.4 Delegating Control

Pending intents are created from intent messages. To create a pending intent, an application defines an intent message and specifies whether it should be used for an activity, service, or broadcast. The pending intent itself is simply a reference to the Intent, and can be shared via RPC to another process within the application, or to another application altogether. When another application receives a pending intent, it can fill in any missing fields (e.g., “extras” values), and invoke the intent. When the intent is invoked, it targets the predefined component type (e.g., activity) and executes within the protection domain of the application that created the pending intent. Frequently, applications use pending intents as “long-term” callbacks, e.g., to be woken up by the system’s alarm service, or Android’s notification manager that allows the user to resume applications based on events.

A vulnerability can result if a pending intent is created from an intent message that does not explicitly define the target component. If the target component is not defined, the application receiving the pending intent can effectively redirect the intent to the application of its choosing. Figure 6.8 shows the FSM for control flow analysis to identify pending intents created from intent messages without a specified target component.

In the figure, state variable `i` tracks an intent message from creation. Once a target component is specified, the FSM transitions to the `targeted` state, which can never transition to the `error` state. However, if a pending intent is created from an intent message with the `empty` state, an error indicating a vulnerability is reported. Here, `$getp()` is a macro that matches the *getActivity()*, *getService()*, and *getBroadcast()* methods of the *PendingIntent* class in the *android.app* namespace.

Note that this FSM does not identify how the pending intent is used. In a practical threat model, sending the pending intent to a system application is not a vulnerability. However, for our analysis, we seek to understand how applications use pending intents

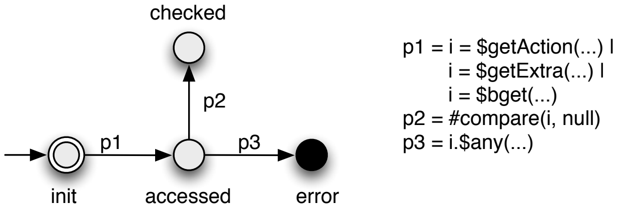

Figure 6.9. FSM for control flow analysis of null checks on IPC input.

in potentially unsafe scenarios. Future work will consider how to reduce the possibility of false positive.

6.4.2.5 Null Checks on IPC Input

Android terminates an application if it dereferences null. Frequently, applications perform actions based on objects received via IPC. If the application does not perform null checks on received objects, remote applications can cause the application to crash. This denial of service vulnerability is particularly useful for an adversary attempting to stop background service functionality, as the user may not be aware the service has terminated.

Figure 6.9 shows the FSM for control flow analysis to identify missed null checks on IPC input. The FSM tracks any variable `i` and transitions to state `accessed` when `i` is input from IPC. Here, `$getAction()` is the similarly named method of the *Intent* class in the *android.content* namespace; `$getExtra()` matches any `get.*Extra()` method of the *Intent* class, and `$bget` matches any `get.*()` method of the *Bundle* class in the *android.os* namespace. The FSM uses the analysis directive `#compare()` to determine if the application compares `i` to `null`. However, if this does not occur, `$any()` matches any method of object `i`, which will result in a null dereference.

To gain additional insight as to where null dereferences on IPC input occur, we define multiple versions of the FSM that are only evaluated in *Activity*, *Service*, and *BroadcastReceiver* classes. Note that this limits analysis to methods defined within classes extending these classes, and does not include null dereferences in objects used by activities, services, and broadcast receivers. Therefore, our analysis considers both general and specific cases.

Table 6.1. Access of Phone Identifier APIs

Identifier	**# Calls**	**# Apps**	**# w/ Permission***
Phone Number	167	129	105
IMEI	378	216	184†
IMSI	38	30	27
ICC-ID	33	21	21
Total Unique	-	246	210†

* Defined as having the `READ_PHONE_STATE` permission.

† Only 1 app did not also have the `INTERNET` permission.

6.5 Application Analysis Results

In this section, we document the program analysis results and manual inspection of identified violations. The automated program analysis took 96.3 hours (about 4 days) of compute time.

6.5.1 Information Misuse

In this section, we explore how sensitive information is being leaked [32, 6] by the studied applications through information sinks including *OutputStream* objects retrieved from *URLConnection*s, HTTP GET and POST parameters in *HttpClient* connections, and the string used for *URL* objects. Future work may also include SMS as a sink.

6.5.1.1 Phone Identifiers

We studied four phone identifiers: phone number, IMEI (device identifier), IMSI (subscriber identifier), and ICC-ID (SIM card serial number). We performed two types of analysis: *a*), we scanned for APIs that access identifiers, and *b*) used data flow analysis to identify code capable of sending the identifiers to the network.

Table 6.1 summarizes APIs calls that receive phone identifiers. In total, 246 applications (22.4%) included code to obtain a phone identifier; however, only 210 of these applications have the `READ_PHONE_STATE` permission required to obtain access (115 applications have this permission, but do not access these APIs). We observe from Table 6.1 that applications most frequently access the IMEI (216 applications, 19.6%). The phone number is used second most (129 applications, 11.7%). Finally, the IMSI and ICC-ID are very rarely used by applications (less than 3%).

Table 6.2 indicates the data flows that exfiltrate phone identifiers. The 33 applications have the `INTERNET` permission, but 1 application does not have the `READ_PHONE_STATE`

Table 6.2. Detected Data Flows to Network Sinks

	Phone Identifiers		**Location Info.**	
Sink	**# Flows**	**# Apps**	**# Flows**	**# Apps**
OutputStream	10	9	0	0
HttpClient Param	24	9	12	4
URL Object	59	19	49	10
Total Unique	-	33	-	13

permission (see developer toolkits in Section 6.5.3). We found data flows for all four identifier types: 25 applications have IMEI data flows; 10 applications have phone number data flows; 5 applications have IMSI data flows; and 4 applications have ICC-ID data flows.

To gain a better understanding of how phone identifiers are used, we manually inspected all 33 identified applications, as well as several additional applications that contain calls to identifier APIs. For all but 1 of the 33 applications, we confirmed exfiltration. While we could not confirm the exact data flow for the remaining application due to code complexity, we identified a data flow not found by program analysis. The analysis informs the following findings.

Finding 1 - *Phone identifiers are frequently leaked through plaintext requests.* Most sinks are HTTP GET or POST parameters. HTTP parameter names for the IMEI include: "uid," "user-id," "imei," "deviceId," "deviceSerialNumber," "devicePrint," "X-DSN," and "uniquely_code"; phone number names include "phone" and "mdn"; and IMSI names include "did" and "imsi." In one case we identified an HTTP parameter for the ICC-ID, but the developer had mislabeled it as "imei."

Finding 2 - *Phone identifiers are used as device fingerprints.* Several data flows directed us towards code that reports not only phone identifiers, but also other phone properties to a remote server. For example, a wallpaper application (com.eoeandroid.eWallpapers.cartoon) contains a class named *SyncDeviceInfosService* that collects the IMEI and attributes such as the OS version and device hardware. The method *sendDeviceInfos()* sends this information to a server. In another application (com.avantar.wny), the method *PhoneStats.toUrlFormatedString()* creates a URL parameter string containing the IMEI, device model, platform, and application name. While the intent is not clear, such fingerprinting indicates that phone identifiers are used for more than a unique identifier.

Finding 3 - *Phone identifiers, specifically the IMEI, are used to track individual users.* Several applications contain code that binds the IMEI as a unique identifier to network requests. For example, some applications (e.g. com.Qunar and com.nextmobileweb.craigsphone) appear to bundle the IMEI in search queries; in a travel application (com.visualit.tubeLondonCity), the method *refreshLiveInfo()* includes the IMEI in a URL; and a "keyring" application (com.froogloid.kring.google.zxing.client.android) appends the IMEI to a variable named *retailerLookupCmd*. We also found functionality that includes the IMEI when checking for updates (e.g., com.webascender.callerid, which also includes the phone number) and retrieving advertisements (see Finding 6). Furthermore, we found two applications (com.taobo.tao and raker.duobao.store) with network access wrapper methods that include the IMEI for all connections. These behaviors indicate that the IMEI is used as a form of "tracking cookie".

Finding 4 - *The IMEI is tied to personally identifiable information (PII).* The common belief that the IMEI to phone owner mapping is not visible outside the cellular network is no longer true. In several cases, we found code that bound the IMEI to account information and other PII. For example, applications (e.g. com.slacker.radio and com.statefarm.pocketagent) include the IMEI in account registration and login requests. In another application (com.amazon.mp3), the method *linkDevice()* includes the IMEI. Code inspection indicated that this method is called when user chooses to "Enter a claim code" to redeem gift cards. We also found IMEI use in application functionality (e.g., com.morbe.guarder and com.fm207.discount) to sending comments or reporting problems. Finally, we found one application (com.andoop.highscore) that appears to bundle the IMEI when submitting high scores for games. Thus, it seems clear that databases containing mappings between physical users and IMEIs are being created.

Finding 5 - *Not all phone identifier use leads to exfiltration.* Several applications that access phone identifiers did not exfiltrate the values. For example, one application (com.amazon.kindle) creates a device fingerprint for a verification check. The fingerprint is kept in "secure storage" and does not appear to leave the phone. Another application (com.match.android.matchmobile) assigns the phone number to a text field used for account registration. While the value is sent to the network during registration, the user is presented an opportunity to change it.

Finding 6 - *Phone identifiers are sent to advertisement and analytics servers.* Many applications have custom ad and analytics functionality. For example, in one application (com.accuweather.android), the class *ACCUWX_AdRequest* is an IMEI data flow sink. Another application (com.amazon.mp3) defines Android service component *AndroidMetricsManager* with an IMEI data flow sink. Phone identifier data flows also occur in ad libraries. For example, we found a phone number data flow in the `com/wooboo/adlib_android` library used by several applications (e.g., cn.ecook, com.superdroid.sqd, and

Table 6.3. Access of Location APIs

Identifier	**# Uses**	**# Apps**	**# w/ Perm.***
getLastKnownLocation	428	204	148
LocationListener	652	469	282
requestLocationUpdates	316	146	128
Total Unique	-	505	304†

* Defined as having a `LOCATION` permission.

† In total, 5 apps did not also have the `INTERNET` permission.

com.superdroid.ewc). Section 6.5.3 discusses ad libraries.

6.5.1.2 Location Information

Location information is accessed in two ways: (1) calling *getLastKnownLocation()*, and (2) defining callbacks in a *LocationListener* object passed to *requestLocationUpdates()*. Due to code recovery failures, not all *LocationListener* objects have corresponding *requestLocationUpdates()* calls. We scanned for all three constructs.

Table 6.3 summarizes the access of location information. In total, 505 applications (45.9%) attempt to access location and only 304 (27.6%) have the permission to do so. The separation between *LocationListener* and *requestLocationUpdates()* is primarily due to the AdMob library, which defined the former but has no calls to the latter.

Table 6.2 shows detected location data flows to the network. To overcome missing code challenges, the data flow source was defined as the *getLatitude()* and *getLongitude()* methods of the *Location* object retrieved from the location APIs. We manually inspected the 13 applications with location data flows. Many data flows appeared to reflect legitimate uses of location for weather, classifieds, points of interest, and social networking services. Inspection of the remaining applications informs the following findings:

Finding 7 - *The granularity of location reporting may not always be obvious to the user.* In one application (com.andoop.highscore) both the city/country *and* geographic coordinates are sent along with high scores. Users may be aware of regional geographic information associated with scores, but it was unclear if users are aware that precise coordinates are also used.

Finding 8 - *Location information is sent to advertisement servers.* Several location data flows appeared to terminate in network connections used to retrieve ads. For example, in two applications (com.avantar.wny and com.avantar.yp) appended the location to the variable *webAdURLString*. Motivated by [6], we inspected the AdMob library to determine why no data flow was found and determined that source code recovery failures led to the false negatives. Section 6.5.3 expands on ad and analytics libraries.

6.5.2 Phone Misuse

This section explores misuse of the smartphone interfaces. We begin by investigating voice and text use, followed by on-phone audio and video, use of data interfaces, and access to the list of installed applications.

6.5.2.1 Telephony Services

Smartphone malware can provide the author direct compensation by making phone calls or sending SMS messages to premium-rate numbers [25, 20]. We defined queries to identify such malicious behavior: (1) a constant used for the SMS destination number; (2) creation of *URI* objects with a "`tel:`" prefix (used for phone call intent messages) and the string "900" (a premium-rate number prefix in the US); and (3) any *URI* objects with a "`tel:`" prefix. The analysis informs the following findings.

Finding 9 - *Applications do not appear to be using fixed phone number services.* We found zero applications using a constant destination number for the SMS API. Note that our analysis specification is limited to constants passed directly to the API and *final* variables, and therefore may have false negatives. We found two applications creating *URI* objects with the "`tel:`" prefix and containing the string "900". One application included code to call "`tel://0900-9292`", which is a premium-rate number (€0.70 per minute) for travel advice in the Netherlands. However, this did not appear malicious, as the application (com.Planner9292) is designed to provide travel advice. The other application contained several hard-coded numbers with "900" in the last four digits of the number. The SMS and premium-rate analysis results are promising indicators for non-existence of malicious behavior. Future analysis should consider more premium-rate prefixes.

Finding 10 - *Applications do not appear to be misusing voice services.* We found 468 *URI* objects with the "`tel:`" prefix in 358 applications. We manually inspected a sample of applications to better understand phone number use. We found: (1) applications frequently include phone call functionality for customer service; (2) the "`CALL`" and "`DIAL`" intent actions were used equally for the same purpose (`CALL` calls immediately and requires the `CALL_PHONE` permission, whereas for `DIAL` the user confirms the call in the dialer and requires no permission); (3) not all hard-coded telephone numbers are used to make phone calls, e.g., the AdMob had a apparently unused phone number hard coded.

6.5.2.2 Background Audio/Video

Microphone and camera eavesdropping on smartphones is a real concern [226]. We analyzed application eavesdropping behaviors, specifically: (1) recording video without calling *setPreviewDisplay()* (this API is always required for still image capture); (2) *AudioRecord.read()* in code not reachable from an Android activity component; and (3) *MediaRecorder.start()* in code not reachable from an activity component.

Finding 11 - *Applications do not appear to be misusing video recording.* We found no applications that record video without calling *setPreviewDisplay()*. The query reasonably did not consider the value passed to the preview display, and therefore may create false negatives. For example, the "preview display" might be one pixel in size. The *MediaRecorder.start()* query detects audio recording, but it also detects video recording. This query found two applications using video in code not reachable from an activity; however the classes extended *SurfaceView*, which is used by *setPreviewDisplay()*.

Finding 12 - *Applications do not appear to be misusing audio recording.* We found eight uses in seven applications of *AudioRecord.read()* without a control flow path to an activity component. Of these applications, three provide VoIP functionality, two are games that repeat what the user says, and one provides voice search. In these applications, audio recording is expected; the lack of reachability was likely due to code recovery failures. The remaining application did not have the required `RECORD_AUDIO` permission and the code most likely was part of a developer toolkit. The *MediaRecorder.start()* query identified an additional five applications recording audio without reachability to an activity. Three of these applications have legitimate reasons to record audio: voice search, game interaction, and VoIP. Finally, two games included audio recording in a developer toolkit, but no record permission, which explains the lack of reachability. Section 6.5.3.2 discusses developer toolkits.

6.5.2.3 Socket API Use

Java sockets represent an open interface to external services, and thus are a potential source of malicious behavior. For example, smartphone-based botnets have been found to exist on "jailbroken" iPhones [216]. We observe that most Internet-based smartphone applications are HTTP clients. Android includes useful classes (e.g., *HttpURLConnection* and *HttpClient*) for communicating with Web servers. Therefore, we queried for applications that make network connections using the *Socket* class.

Finding 13 - *A small number of applications include code that uses the* Socket *class directly.* We found 177 *Socket* connections in 75 applications (6.8%). Many applications are flagged for inclusion of well-known network libraries such as `org/apache/thrift`, `org/apache/commons`, and `org/eclipse/jetty`, which use sockets directly. Socket factories were also detected. Identified factory names such as *TrustAllSSLSocketFactory*, *AllTrustSSLSocketFactory*, and *NonValidatingSSLSocketFactory* are interesting as potential vulnerabilities, but we found no evidence of malicious use. Several applications also included their own HTTP wrapper methods that duplicate functionality in the Android libraries, but did not appear malicious. Among the applications including custom network connection wrappers is a group of applications in the "Finance" category implementing cryptographic network protocols (e.g., in the `com/lumensoft/ks` library). We also note that all of these applications use Asian character sets for their market descriptions, therefore we could not determine the exact purpose.

Finding 14 - *We found no evidence of malicious behavior by applications using* Socket *directly.* We manually inspected all 75 applications to determine if *Socket* use seemed appropriate based on the application description. Our survey yielded a diverse array of *Socket* uses, including: file transfer protocols, chat protocols, audio and video streaming, and network connection tethering, among other uses excluded for brevity. A handful of applications have socket connections to hard-coded IP address and non-standard ports. For example, one application (com.eingrad.vintagecomicdroid) downloads comics from 208.94.242.218 on port 2009. Additionally, two of the aforementioned financial applications (com.miraeasset.mstock and kvp.jjy.MispAndroid320) include the `kr/co/shiftworks` library that connects to 221.143.48.118 on port 9001. Furthermore, one application (com.tf1.lci) connects to 209.85.227.147 on port 80 within a class named *AdService* and subsequently calls *getLocalAddress()* to retrieve the phone's IP address. Overall, we found no evidence of malicious behavior, but a more in-depth investigation is warranted for several applications.

6.5.2.4 Installed Applications

The list of installed applications provides valuable marketing data. Android has two relevant APIs types: (1) a set of *get* APIs returning the list of installed applications or package names; and (2) a set of *query* APIs that mirrors Android's runtime intent resolution, but can be made generic. We found 54 uses of the get APIs in 45 applications, and 1015 uses of the query APIs in 361 applications. Sampling these applications, we found the following.

Finding 15 - *Applications do not appear to be harvesting information about which applications are installed on the phone.* In all but two cases, the sampled applications using the get APIs search the results for a specific application. One application (com.davidgoemans.simpleClockWidget) defines a method that returns the list of all installed applications, but the results were only displayed to the user. The second application (raker.duobao.store) defines a similar method, but it only appears to be called by unused debugging code. Our survey of the query APIs identified three calls within the AdMob library duplicated in many applications. These uses queried specific functionality and thus are not likely to harvest application information. The one non-AdMob application we inspected queried for specific functionality, e.g., speech recognition, and thus did not appear to attempt harvesting.

6.5.3 Included Libraries

Libraries included by applications are often easy to identify due to namespace conventions: i.e., the source code for com.foo.appname typically exists in `com/foo/appname`. During our manual inspection, we documented advertisement and analytics library paths. We also found applications sharing "developer toolkits," i.e., a common set of developer utilities.

6.5.3.1 Advertisement and Analytics Libraries

We identified 22 library paths containing ad or analytics functionality. Sampled applications frequently contained multiple of these libraries. Using the paths listed in Table 6.4, we found: 1 app has 8 libraries; 10 apps have 7 libraries; 8 apps have 6 libraries; 15 apps have 5 libraries; 37 apps have 4 libraries; 32 apps have 3 libraries; 91 apps have 2 libraries; and 367 apps have 1 library.

Table 6.4 shows advertisement and analytics library use. In total, at least 561 applications (51%) include these libraries; however, additional libraries may exist, and some applications include custom ad and analytics functionality. The AdMob library is used most pervasively, existing in 320 applications (29.1%). Google Ads is used by 206 applications (18.7%). We observe from Table 6.4 that only a handful of libraries are used pervasively.

Several libraries access phone identifier and location APIs. Given the library purpose, it is easy to speculate data flows to network APIs. However, many of these flows were not

Table 6.4. Identified Ad and Analytics Library Paths

Library Path	**# Apps**	**Format**	**Obtains**[*]
com/admob/android/ads	320	Obfuscated	L
com/google/ads	206	Plain	-
com/flurry/android	98	Obfuscated	-
com/qwapi/adclient/android	74	Plain	L, P, E
com/google/android/apps/analytics	67	Plain	-
com/adwhirl	60	Plain	L
com/mobclix/android/sdk	58	Plain	L, E[‡]
com/millennialmedia/android	52	Plain	-
com/zestadz/android	10	Plain	-
com/admarvel/android/ads	8	Plain	-
com/estsoft/adlocal	8	Plain	L
com/adfonic/android	5	Obfuscated	-
com/vdroid/ads	5	Obfuscated	L, E
com/greystripe/android/sdk	4	Obfuscated	E
com/medialets	4	Obfuscated	L
com/wooboo/adlib_android	4	Obfuscated	L, P, I[†]
com/adserver/adview	3	Obfuscated	L
com/tapjoy	3	Plain	-
com/inmobi/androidsdk	2	Plain	E[‡]
com/apegroup/ad	1	Plain	-
com/casee/adsdk	1	Plain	S
com/webtrends/mobile	1	Plain	L, E, S, I
Total Unique Apps	561	-	-

[*] L = Location; P = Phone number; E = IMEI; S = IMSI; I = ICC-ID

[†] In 1 app, the library included “L”, while the other 3 included “P, I”.

[‡] Direct API use not decompiled, but wrapper *.getDeviceId()* called.

detected by program analysis. This is (likely) a result of code recovery failures and flows through Android IPC. For example, AdMob has known location to network data flows [6], and we identified a code recovery failure for the class implementing that functionality. Several libraries are also obfuscated, as mentioned in Section 6.6. Interesting, 6 of the 13 libraries accessing sensitive information are obfuscated. The analysis informs the following additional findings.

Finding 16 - *Ad and analytics library use of phone identifiers and location is sometimes configurable.* The `com/webtrends/mobile` analytics library (used by com.statefarm.pocketagent), defines the *WebtrendsIdMethod* class specifying four identifier types. Only one type, “*system_id_extended*” uses phone identifiers (IMEI, IMSI, and ICC-ID). It is unclear which identifier type was used by the application. Other libraries provide similar configuration.

For example, the AdMob SDK documentation [228] indicates that location information is only included if a package manifest configuration enables it.

Finding 17 - *Analytics library reporting frequency is often configurable.* During manual inspection, we encountered one application (com.handmark.mpp.news.reuters) in which the phone number is passed to *FlurryAgent.onEvent()* as generic data. This method is called throughout the application, specifying event labels such as "GetMoreStories," "StoryClickedFromList," and "ImageZoom." Here, we observe the main application code not only specifies the phone number to be reported, but also report frequency.

Finding 18 - *Advertisement and analytics libraries probe for permissions.* The `com/webtrends/mobile` library accesses the IMEI, IMSI, ICC-ID, and location. The (*WebtrendsAndroidValueFetcher*) class uses try/catch blocks to probe, a *SecurityException* is thrown when an application does not have the proper permission. Similar functionality exists in the `com/casee/adsdk` library (used by com.fish.luny). In *AdFetcher.getDeviceId()*, Android's *checkCallingOrSelfPermission()* method is evaluated before accessing the IMSI.

6.5.3.2 Developer Toolkits

Finding 19 - *Some developer toolkits replicate dangerous functionality.* Three wallpaper applications by "callmejack" (com.eoeandroid.eWallpapers.cartoon, com.jackeey.wallpapers.all1.orange, and com.jackeey.eWallpapers.gundam) include utilities in the library path `com/jackeeywu/apps/eWallpaper`. This library has data flow sinks for the phone number, IMEI, IMSI, and ICC-ID. In July 2010, Lookout, Inc. reported a wallpaper application by developer "jackeey,wallpaper" as sending these identifiers to `imnet.us` [34]. This report also indicated that the developer changed his name to "callmejack". While the original "jackeey,wallpaper" application was removed from the Android Market, the applications developed by "callmejack" remained as of September 2010.[1]

Finding 20 - *Some developer toolkits probe for permissions.* In one application (com.july.cbssports.activity), we found code in the `com/julysystems` library that evaluates Android's *checkPermission()* method for the `READ_PHONE_STATE` and `ACCESS_FINE_LOCATION` permissions before accessing the IMEI, phone number, and last known location, respectively. A second application (v00032.com.wordplayer) defines the *CustomExceptionHander* class to send an exception event to an HTTP URL. The class attempts to retrieve the phone number within a try/catch block, catching a generic *Exception*.

---

[1] Fortunately, these dangerous applications are now nonfunctional, as the `imnet.us` NS entry is `NS1.SUSPENDED-FOR.SPAM-AND-ABUSE.COM`.

The application does not have the `READ_PHONE_STATE` permission, therefore, this class is likely a developer toolkit.

Finding 21 - *Well-known brands sometimes commission developers that include dangerous functionality.* The `com/julysystems` developer toolkit identified as probing for permissions appears in two applications with reputable names listed in the Android Market. “CBS Sports Pro Football” (com.july.cbssports.activity) is provided by “CBS Interactive, Inc.”, and “Univision Fütbol” (com.july.univision) is provided by “Univision Interactive Media, Inc.”. As both applications have location and phone state permissions, they potentially misuse information.

Similarly, “USA TODAY” (com.usatoday.android.news) provided by “USA TODAY” and “FOX News” (com.foxnews.android) provided by “FOX News Network, LLC” contain the `com/mercuryintermedia` toolkit. Both applications contain an Android activity component named *MainActivity*. In the initialization phase, the IMEI is retrieved and passed to *ProductConfiguration.initialize()* (part of the `com/mecuryintermedia` toolkit). Both applications have IMEI to network data flows through this method.

6.5.4 Android-specific Vulnerabilities

Android provides developers a flexible programming interface. Vulnerabilities can result from API misuse. We looked for several types of vulnerabilities based on best secure coding practices for Android [41, 227].

6.5.4.1 Leaking Information to Logs

Android provides centralized logging via the *Log* API, which can displayed with the “`logcat`” command. While `logcat` is a debugging tool, applications with the `READ_LOGS` permission can read these log messages. The Android documentation for this permission indicates that “[the logs] can contain slightly private information about what is happening on the device, but should never contain the user’s private information.” We looked for data flows from phone identifier and location APIs to the Android logging interface and found the following.

Finding 22 - *Private information is written to Android’s general logging interface.* We found 253 data flows in 96 applications for location information, and 123 flows in 90 applications for phone identifiers. Frequently, URLs containing this private information are logged just before a network connection is made. Therefore, applications with the `READ_LOGS` permission can access private information.

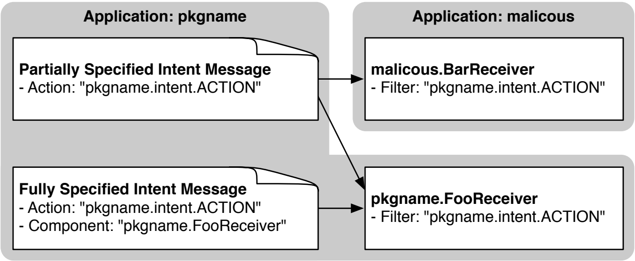
Application: pkgname
Application: malicous
Partially Specified Intent Message
- Action: "pkgname.intent.ACTION"
malicous.BarReceiver
- Filter: "pkgname.intent.ACTION"
Fully Specified Intent Message
- Action: "pkgname.intent.ACTION"
- Component: "pkgname.FooReceiver"
pkgname.FooReceiver
- Filter: "pkgname.intent.ACTION"

Figure 6.10. Eavesdropping on unprotected intent messages.

6.5.4.2 Leaking Information via IPC

As depicted in Figure 6.10, any application can receive intent broadcasts that do not specify the target component or protect the broadcast with a permission (permission variant not shown). This is unsafe if the intent contains sensitive information. We found 271 such unsafe intent broadcasts with extras data in 92 applications (8.4%). Sampling these applications, we found several intents matching this property used to install shortcuts to the home screen.

Finding 23 - *Applications broadcast private information in IPC accessible to all applications.* We found many cases of applications sending unsafe intents to action strings containing the application's namespace (e.g., "pkgname.intent.ACTION" for application pkgname). The contents of the bundled information varied. In some instances, the data was not sensitive, e.g., widget and task identifiers. However, we also found sensitive information. For example one application (com.ulocate) broadcasts the user's location to the "com.ulocate.service.LOCATION" intent type without protection. Another application (com.himsn) broadcasts the instant messaging client's status to the "cm.mz.stS" action string. This confirms that malicious applications can eavesdrop on sensitive information in IPC, and in some cases, gain access to information that requires a permission (e.g., location).

6.5.4.3 Unprotected Broadcast Receivers

Applications use broadcast receiver components to receive intent messages. Broadcast receivers define "intent filters" to subscribe to specific event types are public. If the receiver is not protected by a permission, a malicious application can forge messages.

Finding 24 - *Few applications are vulnerable to forging attacks to dynamic broadcast receivers.* We found 406 unprotected broadcast receivers in 154 applications (14%). We found an large number of receivers subscribed to system defined intent types. Android introduced “protected broadcasts” for system intent types to eliminate forging [5]. We found one application with an unprotected broadcast receiver for a custom intent type; however this case appears to have limited impact. Additional sampling may uncover more cases.

6.5.4.4 Intent Injection Attacks

Intent messages are also used to start activity and service components. An intent injection attack occurs if the intent address is derived from untrusted input. We found 10 data flows from the network to an intent address in 1 application. We could not confirm the data flow and classify it a false positive. The data flow sink exists in a class named *ProgressBroadcastingFileInputStream*. No decompiled code references this class, and all data flow sources are calls to *URLConnection.getInputStream()*, which is used to create *InputStreamReader* objects. We believe the false positives results from the program analysis modeling of classes extending *InputStream*.

We found 80 data flows from IPC to an intent address in 37 applications. We classified the data flows by the sink: the *Intent* constructor is the sink for 13 applications; *setAction()* is the sink for 16 applications; and *setComponent()* is the sink for 8 applications. These sets are disjoint. Of the 37 applications, we found that 17 applications set the target component class explicitly (all except 3 use the *setAction()* data flow sink), e.g., to relay the action string from a broadcast receiver to a service. We also found four false positives due to our assumption that all *Intent* objects come from IPC (a few exceptions exist). For the remaining 16 cases, the analysis informs the following.

Finding 25 - *Some applications define intent addresses based on IPC input.* Three applications use IPC input strings to specify the package and component names for the *setComponent()* data flow sink. Similarly, one application uses the IPC “extras” input to specify an action to an *Intent* constructor. Two additional applications start an activity based on the action string returned as a result from a previously started activity. However, to exploit this vulnerability, the applications must first start a malicious activity. In the remaining cases, the action string used to start a component is copied directly into a new intent object. This can be exploited by specifying the vulnerable component’s name directly and controlling the action string.

6.5.4.5 Delegating Control

Applications can delegate actions to other applications using a “pending intent.” An application first creates an intent message as if it was performing the action. It then creates a reference to the intent based on the target component type (restricting how it can be used). The pending intent recipient cannot change values, but it can fill in missing fields. Therefore, if the intent address is unspecified, the remote application can redirect an action that will be performed with the original application's permissions.

Finding 26 - *Few applications unsafely delegate actions.* We found 300 unsafe pending intent objects in 116 applications (10.5%). Sampling these applications, we found an overwhelming number of pending intents used for either: (1) Android's UI notification service; (2) Android's alarm service; or (3) communicating between a UI widget and the main application. None of these cases allow manipulation by a malicious application. We found two applications that send unsafe pending intents via IPC. However, exploiting these vulnerabilities appears to provides negligible adversarial advantage.

6.5.4.6 Null Checks on IPC Input

Android applications frequently process information from intent messages received from other applications. Null dereferences cause an application to crash, and can thus be used to as a denial of service.

Finding 27 - *Applications frequently do not perform null checks on IPC input.* We found 3925 potential null dereferences on IPC input in 591 applications (53.7%). Most occur in classes for activity components (2,484 dereferences in 481 applications). Null dereferences in activity components have minimal impact, as the application crash is obvious to the user. We found 746 potential null dereferences in 230 applications within classes defining broadcast receiver components. Applications commonly use broadcast receivers to start background services, therefore it is unclear what effect a null dereference in a broadcast receiver will have. Finally, we found 72 potential null dereferences in 36 applications within classes defining service components. Applications crashes corresponding to these null dereferences have a higher probability of going unnoticed. The remaining potential null dereferences are not easily associated with a component type.

6.5.4.7 SDcard Use

Any application that has access to read or write data on the SDcard can read or write any other application's data on the SDcard. We found 657 references to the SDcard in 251 applications (22.8%). Sampling these applications, we found a few unexpected uses. For example, the `com/tapjoy` ad library (used by com.jnj.mocospace.android) determines the free space available on the SDcard. Another application (com.rent) obtains a URL from a file named `connRentInfo.dat` at the root of the SDcard.

6.5.4.8 JNI Use

Applications can include functionality in native libraries using the Java Native Interface (JNI). As these methods are not written in Java, they have inherent dangers. We found 2762 calls to native methods in 69 applications (6.3%). Investigating the application package files, we found that 71 applications contain `.so` files. This indicates two applications with an `.so` file either do not call any native methods, or the code calling the native methods was not decompiled. Across these 71 applications, we found 95 included `.so` files, 82 of which have unique names.

6.5.5 General Application Vulnerabilities

We analyzed the application source code for general Java application vulnerabilities based on industry-standard criteria [225]. Many of the criteria are irrelevant due to either artifacts of the decompilation process (e.g., variables including the "$" character) or clearly irrelevant to Android (e.g., J2EE vulnerabilities). For the applicable criteria, we identified 5,325 issues. A breakdown of the number of instances and affected applications is shown in Table 6.5. We observe that analysis results are unevenly distributed. Only 564 out of 1,100 application had potential vulnerabilities detected by the general criteria. Further, less than a third of the 564 applications—16.91% of the 1,100—account for over 82% of the detected code locations.

6.5.5.1 Password Misuse

Finding 28 - *Few applications have hard-coded or empty passwords.* We only found eight applications with hard-coded passwords. Two of these applications rely on a Twitter and an ICQ login. In a library common to two "Finance" applications, unique username and password pairs are specified to authenticate to the same hard-coded IP. The purpose of this code was unclear. The fifth application uses a hard-coded password to authenticate custom error reporting. The sixth application contained a hard-coded password for a demo account. Finally, in the last two applications, the same GMail username and password is used in a class named *SendTest*. All of the other sampled code locations for hard-coded or empty passwords are initializers (e.g., "changeit") in library code.

Finding 29 - *We found no evidence of plaintext passwords written to file.* All of the sampled code locations identified as writing plaintext passwords to file are false positives. Most studied code locations exist in unused library code (e.g., obtaining a password from *stdin.readline()*, which cannot be used in Android). Note that the Wells Fargo application recently identified as writing a plaintext password to file [229] did not appear to have this functionality in the recovered source code.

6.5.5.2 Cryptography Misuse

Finding 30 - *Some applications use unsafe cryptographic keys and algorithms.* We found 7 applications including the toolkit class *SecurityUtil* that uses a variable containing the "device ID" (most likely the IMEI) as a DES encryption key. If the device ID is not available, a hard-coded constant is used. We found several other uses of DES; however these are in the NTLM implementation of an Apache library. It is unclear if this functionality is used.

Finding 31 - *Few applications use insufficient key size.* We found two applications (com.scfirstbank and edaily.daishin) using RSA with a 1024-bit modulus. A third application (com.slacker.radio) uses RSA with a 512-bit modulus. Several additional instances exist in libraries.

Finding 32 - *We found no evidence of PRNG misuse.* Insecure randomness accounts for 1,681 of the 2,278 flags in the cryptography category in Table 6.5. Our sampling found a library for one application (com.scfirstbank) where a non-cryptographic PRNG is selected only if the requested cryptographic one is not available; however the method did not appear to be used. Finally, the analysis looks for all PRNG uses, most of which are not cryptographically related (e.g., selecting keywords for ad targeting).

6.5.5.3 Injection Vulnerabilities

Finding 33 - *Few applications have path manipulation vulnerabilities.* Path manipulation consisted of 1,228 of the 2,520 flagged code locations in the injection category. Most of the sampled cases existed within a *main()* method of a library (e.g., for Base64 encoding/decoding); however, Android does not execute *main()* methods. Android programming conventions also led to false positives. For example, content provider components that share files commonly store the filename in an SQLite database. This filename was detected as untrusted. We only found one application with a path manipulation vulnerability. In this case, a filename is based on values received from an intent message; however, the conditions under which the vulnerability can be exploited are unclear.

Finding 34 - *We found no evidence of database or command injection vulnerabilities.* All sampled flags for database injection are false positives. We speculate the non-existence of this vulnerability is the widespread use of parameterized SQL queries in Android.

6.6 Study Limitations

The study described in the previous section was limited in three ways: *a*) the studied applications were selected with a bias towards popularity; *b*) the program analysis tool cannot compute data and control flows for IPC between components; and *c*) source code recovery failures interrupt data and control flows. Missing data and control flows may lead to false negatives. In addition to the recovery failures, the program analysis tool could not parse 8,042 classes, reducing coverage to 91.34% of the classes.

Additionally, a portion of the recovered source code was obfuscated before distribution. Code obfuscation significantly impedes manual inspection. It likely exists to protect intellectual property, as Google suggests obfuscation using ProGuard (`proguard.sf.net`) for applications using its licensing service [230]. ProGuard primarily protects against readability. It does not obfuscate control flow, therefore it has limited impact for a program analysis tool.

Many forms of obfuscated code are easily recognizable: e.g., class, method, and field names are converted to single letters, producing single letter Java filenames (e.g., `a.java`). For a rough estimate on the use of obfuscation, we searched applications containing `a.java`. In total, 396 of the 1,100 applications contain this file. As discussed in Section 6.5.3, several advertisement and analytics libraries are obfuscated. To obtain a closer estimate of the number of applications whose main code is obfuscated, we searched for `a.java` within a file path equivalent to the package name (e.g., `com/foo/appname` for com.foo.appname). Only 20 applications (1.8%) have this obfuscation property, which is expected for free applications (as opposed to paid applications). However, we stress that the `a.java` heuristic is not intended to be a firm characterization of the percentage of obfuscated code, but rather a means of acquiring insight.

6.7 Summary of Findings

Identifying a singular take-away from a broad study such as this is non-obvious. We come away from the study with two central thoughts; one having to do with the study apparatus, and the other to do with the applications themselves.

`ded` and the program analysis specifications are enabling technologies that open a new door for application certification. We found the approach rather effective despite existing limitations. In addition to further studies of this kind, we see the potential to integrate these tools into an application certification process. We leave such discussions for future work, noting that such integration is challenging for both logistical and technical reasons [2].

On a technical level, we found the security characteristics of the top 1,100 free popular applications to be consistent with smaller studies (e.g., Enck et al. [6]). Our findings indicate an overwhelming concern for misuse of privacy sensitive information such as phone identifiers and location information. One might speculate this is because it is difficult to assign malicious intent to such activities. In addition to the existence the information misuse, our manual source code inspection sheds more light on *how* information is misused. We found phone identifiers, e.g., phone number, IMEI, IMSI, and ICC-ID, were used for everything from "cookie-esque" tracking to account numbers. Our findings also support the existence of databases external to cellular providers that link identifiers such as the IMEI to personally identifiable information. Our analysis also identified significant penetration of ad and analytic libraries, occurring in 51% of the studied applications. While this might not be surprising for free applications, the number of ad and analytics libraries included per application was unexpected. One application included as many as eight different libraries. One might question the need for more than one ad and one analytics library. From a vulnerability perspective, we found that many developers fail to take necessary security precautions. For example, sensitive information is frequently written to Android's centralized logs, as well as occasionally broadcast to unprotected IPC. We also identified the potential for IPC injection attacks; however, no cases were readily exploitable.

Finally, our study only characterized one edge of the application space. While we found no evidence of telephony misuse, background recording of audio or video, or abusive network connections, one might argue that such malicious functionality is less likely to occur in popular applications. We focused our study on popular applications to characterize those most frequently used. Future studies should take samples that span application popularity. However, even these samples may miss the existence of truly malicious applications. Future studies should also consider several additional attacks, including installing new applications [231], JNI execution [17], address book exfiltration, destruction of SDcard contents, and phishing [19].

Table 6.5. Source code analysis results for general application vulnerabilities.

Application Category	**Lines of Code**	**Password Mgmt**		**Cryptography**		**Injection**		**All Flag Types**	
		# Flags	**# Apps**	**# Flags**	**# Apps**	**# Flags**	**# Apps**	**# Flags**	**# Apps**
Comics	415,625	0	0	32	10	37	5	69	13
Communication	1,832,514	96	16	119	26	608	26	823	34
Demo	830,471	2	2	21	7	10	2	33	7
Entertainment	709,915	6	3	55	16	90	11	151	22
Finance	709,915	105	8	96	21	91	12	292	26
Games (Arcade)	766,045	2	1	390	27	1	1	393	27
Games (Puzzle)	727,642	1	1	124	31	17	7	142	35
Games (Casino)	985,423	15	3	111	36	6	3	132	38
Games (Casual)	681,429	3	2	129	32	6	4	138	32
Health	847,511	37	11	196	21	85	6	318	26
Lifestyle	778,446	4	2	70	15	85	12	159	22
Multimedia	1,323,805	77	4	186	26	335	23	598	32
News/Weather	1,123,674	4	2	55	15	62	11	121	20
Productivity	1,443,600	26	7	108	18	140	21	274	31
Reference	887,794	4	1	93	25	42	12	139	30
Shopping	1,371,351	14	4	85	23	111	13	210	27
Social	2,048,177	59	12	155	37	478	29	692	41
Libraries	182,655	3	2	9	5	110	11	122	13
Sports	651,881	1	1	17	7	31	11	49	18
Themes	310,203	0	0	95	23	9	4	104	24
Tools	839,866	3	2	56	17	86	20	145	26
Travel	1,419,783	65	8	76	14	80	12	221	20
TOTAL	**21,734,202**	**527**	**92**	**2,278**	**452**	**2,520**	**259**	**5,325**	**564**

Chapter 7

Directions for Smartphone Security

Many millions of consumers rely on smartphones for business and personal needs. While this relatively new technology has changed the way we live our lives, it opens us new and previously unconsidered security risk. Smartphones collect many types of security and privacy sensitive information simply as a matter of operation. All of this information is stored on a small and easy to lose device that always travels with the user, drifts seamlessly between "unknown" wireless networks, and runs applications from largely unknown developers.

Similar commodity desktop platforms, antivirus software has emerged for smartphones. While antivirus software has become an important component of defense in depth on desktops, their usefulness on smartphones is left into question. In fact, in a mailing list discussion, an Android platform engineer called into question what existing Android antivirus applications are actually doing [232]. The application sandboxed environments in platforms such as Android and iOS severely limit traditional antivirus functionality. Furthermore, antivirus software is traditionally a reactive measure to known malware. Similar functionality is provided by application market "kill switches," without wasting energy scanning files.

Several efforts related to antivirus software have also emerged. WhatApp [201] and services such as Lookout's App Genome project [200] seek to inform users of security and privacy risks in applications. These firms have begun monitoring applications market security hygiene not only for Trojans and viruses, but also applications that negatively impact user privacy. In the current smartphone environment, privacy threats are not universally considered malicious, and therefore do not warrant removal by application markets. Third party security services such as WhatApp and Lookout fill a valuable void in the effort to protect end users.

The approaches and techniques described in this dissertation meaningfully advance the technology available to security firms seeking to monitor smartphone application market security hygiene. For example, Lookout uses a permission-based approach similar to that previously published in our Kirin work [5]. Additionally, TaintDroid provides an automated means of identifying privacy risks in applications. Finally, combined with decompilation tools such as `ded`, the source code analysis queries described in Chapter 6 can be applied and extended to quickly identify security failures.

7.1 Host Security: A Conflict of Requirements

Each of our studies of smartphone applications presented evidence of a fundamental conflict between functional requirements and security goals. A primary example of this conflict involves the use of geographic location. Location-aware functionality is a distinct and very valuable feature of mobile operating systems. One can argue that it is a major contributor to the success of smartphones in general. However, geographic location is also fundamentally a privacy sensitive value [33]. Historically, OS protection models ensure the secrecy of sensitive values by restricting access by network-connected processes to a limited set of trusted applications. However, in the smartphone and mobile OS environment, untrusted applications often send location information to the Internet to meet functional requirements. While users often knowingly accept and rely on location information being sent to the Internet, a difficulty arises when determining when network disclosure is desired, and when it is not. Our studies indicate that undesirable network disclosure occurs more frequently than one might expect.

The smartphone community must acknowledge the reality that untrusted applications will sometimes acquire privacy sensitive values. However, this does not mean that users must freely give up privacy. As is common with problems of this nature, a combination of non-technical and technical solutions will help resolve the security dilemma.

7.1.1 Informed Consent

From a non-technical perspective, applications should acquire *informed consent* before accessing privacy sensitive values such as location and phone identifiers. While this can include user interface notifications to indicate, for example, “sending coordinates to determine your location,” it should also include a means of conveying to the user all possible scenarios in which the values will be used. For example, if network server stores geographic coordinates with an anonymous identifier, and the database of coordinates is sold to a marketing firm interested in demographics, this use should be clearly conveyed to the user. Such information is commonly conveyed in a EULA or terms of use displayed on first use of an application. However, it should be noted that the limited screens on smartphones make comprehending lengthy EULAs even more difficult than on PCs.

Future work should study the best way of conveying this information to users, as well as means of encouraging developers to disclose activities. A potential approach is incorporate uniform notifications for applications using privacy sensitive values. In such a model, developers using sensitive values but not reporting how the values are used may be seen as less reputable. However, the effectiveness of such an approach and potential drawbacks is unclear, and hence requires deep investigation.

7.1.2 Privilege Separation

While some applications must send privacy sensitive values to the Internet to meet functional requirements, many applications do so as a result of functionality orthogonal to the application's purpose. In particular, we found location information and phone identifiers are sometimes sent to advertisement servers. Currently, advertisement and analytics functionality is added to applications though the inclusion of libraries. Application developers often add these libraries without knowledge of the functionality contained therein. Furthermore, when performing security analysis on an application, it is difficult to determine what part of the application is responsible for offending behavior.

While smartphone applications are already rather vertical in implemented functionality, it is both technically possible and arguably necessary to provide further *privilege separation*. There is no technical requirement that advertisement and analytics functionality be entirely implemented within an application. For example, Android's modular environment permits the creation of advertisement and analytics services that are installed as separate applications. This approach has several distinct advantages. First, it can remove ambiguity when applications request permissions. Second, it gives OS level protection mechanisms greater context for policy enforcement. Third, it simplifies security analysis of applications and allows greater scrutiny to be devoted towards such service applications. Finally, it promotes transparency and attributes responsibility to arguably dangerous behavior. For example, users can potentially configure an advertisement service application not to use location, as opposed to leaving that decision up to the application developer or provider of the advertisement library.

7.2 Future Work

This dissertation has primarily discussed the security analysis of applications. However, the motivation for the security analysis is to better understand the limitations of existing mobile OS security frameworks, and to inform future enhancements. We now discuss future work in both application analysis and security enhancements for mobile operating systems. We begin with application analysis.

7.2.1 Application Analysis

7.2.1.1 Analysis of Native Libraries

In Chapters 5 and 6, we limited our analysis to the Java portion of Android applications. As the Android Market has matured, popularity and support for native libraries has grown. Games have been a primary motivator, both in order to achieve better performance and to simplify development and maintenance of game engines. For example, the version 2.3 release of Android included enhancements to the Native Developer Kit (NDK) fully native applications and sound support [233].

Future work should investigate techniques for security analysis of native libraries on Android. Related techniques have been used to study misuse of privacy sensitive values on the iOS platform [32]. The analysis of Android native libraries, should however, leverage artifacts of the environment. For example, native libraries may require non-standard conventions to access location and phone identifiers. These conventions may be easy to identify with relatively simple binary analysis. Furthermore, the Java/native library barrier may allow analysis tools to quickly identify functionality as suspicious if it occurs within a native library.

7.2.1.2 Study of Least Privilege

The permission-based security model used by mobile operating systems achieves a closer approximation of least privilege than is available in commodity desktop OS environments. However, studies of Android applications [4], including our own, frequently proclaim that Android's permissions are too coarse. This observation has been made without a deep study of the APIs associated with permissions, or the frequency of which those APIs are used. Such an investigation would be valuable to determine whether or not permissions should be subdivided or combined. For example, the `READ_PHONE_STATE` permission is used both to determine if the phone is ringing *and* to retrieve the handset's phone number. These two distinct security-sensitive operations should arguably have distinct permissions. However, too many permissions can confuse users. A study considering API use within and across applications can inform how permissions may be combined.

7.2.1.3 Characterization of Information Sharing

Contrasted to the iOS platform, which isolates applications with sandboxes, the Android platform encourages a modular development approach wherein functionality spans multiple applications. For example, in Chapter 6, we mentioned an application dedicated to reporting high scores for games. In this example, game developers do not need to implement the reporting functionality themselves. Rather, they direct the user to the high scores application in the application market. The Android Market includes various types of utility compartmentalized into individual applications. Other examples include an interface to read bar-codes, centralized storage of credentials, and text to speech (TTS) services.

In the early Android Market, compartmentalization of utility and information sharing was limited. However, now that the platform has gained a strong world-wide backing and the corpus of available applications has matured, this sharing has become more common. Studying information sharing between applications will provide valuable insight into both potential collusion-based attacks, as well as framework restrictions to enhance device security. One such restriction based on information flow control is described below.

7.2.2 Operating System Enhancements

7.2.2.1 Information Flow Control

Android security is currently enforced by non-comparable permissions. For example, the permission to read the address book cannot be compared with the permission to read Web browser bookmarks, access location information, make phone calls, or install applications. Literature has considered transitive flows of information between applications [124, 125]; however, they assume the existence of a lattice based on some ordering of permissions. Attempting to construct such an ordering of non-comparable permissions will lead false positives and ultimately break functionality.

While using existing permissions to enforce transitive information flow policies is not practical, information flow control provides valuable security semantics for mobile devices. Consider, for example, a smartphone used for both business and personal purposes. A user (or other expert) may label applications as `business`, `personal`, `privacy`, or `financial`. An IFC policy can ensure that business and personal information does

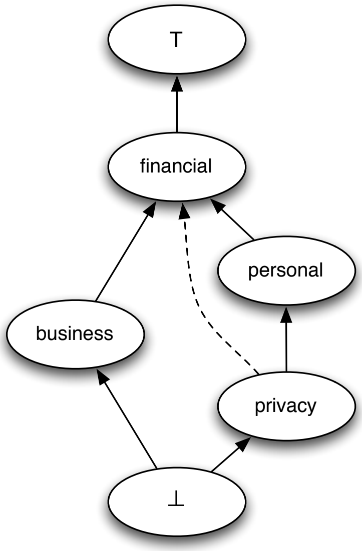

Figure 7.1. Example information flow control lattice for a smartphone.

not mix while allowing both types of applications to contribute to a financial ledger running with the `financial` label. Figure 7.1 encodes such a security goal.

Transitive security goals also exist for specific types of information. Consider, for example, location information. Using information flow semantics, an application can receive location information with the restriction that it does not flow to the Internet, or possibly more realistically, can only flow to a specific remote server based on DNS. Such security goals need not be restricted to well defined information types such as location. It can also be used for application-specific information such as financial data and social networking events. This application-centric approach to IFC policy specification parallels existing decentralized information flow control (DIFC) [70, 71, 72].

A DIFC approach is promising for the Android platform. First, application developers already participate in security policy when assigning and requesting permissions. Second, the component framework lends itself well to the privilege separation required to practically realize DIFC enforcement. Future work will investigate the appropriateness of such information flow control protection models on mobile devices.

7.2.2.2 Maintaining Firmware Integrity

A mobile device's security relies on the integrity of its firmware. Secure and authenticated boot [101] approaches have been discussed for mobile platforms [102, 103, 104, 105]. In fact, Motorola has included a secure boot mechanism in several of its phones [234].

While such precautions are primarily motivated by preventing users from “jailbreaking” devices, a rootkit can just as easily modify the firmware for more malicious purposes.

Smartphones and similar mobile devices have an architectural advantage for maintaining system integrity. Commonly, the device firmware is stored in a read-only partition and is updated only via cryptographically verified procedures. For example, to update an Android phone, the device is booted into a “recovery mode” that verifies the cryptographic signature on an update image before patching the firmware. However, devices can still be jailbroken, because the integrity of the firmware relies on the integrity of the recovery image, which can be compromised if root level privileges are obtained on the device (e.g., the result of a kernel-level privilege escalation attack).

The integrity of the device’s firmware and recovery image can be hardened by reducing the runtime attack surface. Instead of enforcing the read-only protection of the storage partitions in the OS kernel, it should be enforced by hardware. Similar to the way UNIX systems use `setuid` to transition from root to a limited user, the device boot process can set a security bit on storage partitions. When the device boots into normal mode, the read-only bit is set on the firmware and recovery image; however, the partitions remain writable when booting into the recovery image.

7.2.2.3 Extending Protection to the Cloud

Smartphones and similar mobile devices are primarily Internet service devices. However, they also provide a valuable security perimeter for user information. While many times user information is shared with remote servers, this is not always the case. Often, photos, personal notes, and music playlists remain local. However, this is changing. For reasons of consistency between devices and fault tolerance, more and more information is being synchronized and backed up to storage facilities and services on the Internet.

Mobile devices require protection of information synchronized and backed up to Internet-based services. As this information must be available for use in Web browsers and other devices, simple storage encryption approaches are insufficient. Instead, there is a need for a more general framework that decouples storage from application services. Such a framework could protect information until it reaches the user, i.e., only decrypted by a smartphone application or Web browser. To integrate with existing Web applications, the cryptographically protected storage should also be searchable [235]. Future work should investigate the requirements of such a model and determine how best to extend protection of personal information into cloud services.

7.3 Concluding Remarks

Smartphone operating system security is primarily defined by permissions. These permissions are a limited form of capabilities that define the information and resources accessible to applications. This articulation of security policy provides a means of approximating least privilege for applications. While some argue that the permission model is inappropriate for smartphones, because users cannot make informed security decisions, we have shown a possibly more fundamental benefit of permissions. Permissions not only articulate security policy, but they embody a definition of security risk. That is, they document sensitive interfaces and information. This documentation is vital for security analysis, whether it be of applications or the operating system itself.

Our success using permissions to focus security analysis is attributable to properties of smartphone applications. Had developers simply requested all possible permissions, the permission related information would have been meaningless. Fortunately this was not the case. Overall, we found smartphone applications have relatively vertical purpose. Hence, to accomplish its goals, an application only requires use of a limited number of security-sensitive APIs.

Smartphones are the logical conclusion of the Internet's influence on computing technology. More importantly than providing pervasive access to information, they embody the commoditization of discrete services. The vertical purpose nature of applications is a reflection of the desire to market functionality in this way. Increasingly, general purpose computing will follow in these footsteps. As this transition occurs, the approaches and techniques described in this dissertation will become broadly applicable.

Bibliography

[1] Collins, B. (2010), "Word Lens review: the iPhone app that translates whatever you show it," `http://www.pcpro.co.uk/blogs/2010/12/17/word-lens-review-the-iphone-app-that-translates-whatever-you-show-it/`.

[2] McDaniel, P. and W. Enck (2010) "Not So Great Expectations: Why Application Markets Haven't Failed Security," *IEEE Security & Privacy Magazine*, **8**(5), pp. 76–78.

[3] Cannings, R. (2010), "Exercising Our Remote Application Removal Feature," `http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html`.

[4] Barrera, D., H. G. Kayacik, P. C. van Oorshot, and A. Somayaji (2010) "A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android," in *Proceedings of the ACM Conference on Computer and Communications Security.*

[5] Enck, W., M. Ongtang, and P. McDaniel (2009) "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS).*

[6] Enck, W., P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth (2010) "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[7] Enck, W., D. Octeau, S. Chaudhuri, and P. McDaniel (2011) *Path to Android Application Security, Tech. Rep. NAS-TR-0144-2011*, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA.

[8] F-Secure Corporation, "Virus Description: Cabir," `http://www.f-secure.com/v-descs/cabir.shtml`, accessed March 2009.

[9] ——— (2005), "TeliaSonera Finland and F-Secure united against mobile viruses: Cabir spreads at the World Championships," `http://www.f-secure.com/en_EMEA/about-us/pressroom/news/2005/fs_news_20050811_1_eng.html`.

[10] Hyppönen, M. (2007), "Mobile Malware," USENIX Security Symposium, invited Talk.

[11] F-Secure Corporation, "Virus Description: Lasco.A," `http://www.f-secure.com/v-descs/lasco_a.shtml`, accessed March 2009.

[12] ———, "Virus Description: Commwarrior," `http://www.f-secure.com/v-descs/commwarrior.shtml`, accessed March 2009.

[13] ———, "Virus Description: Skulls.A," `http://www.f-secure.com/v-descs/skulls.shtml`, accessed March 2009.

[14] ———, "Virus Description: Drever.A," `http://www.f-secure.com/v-descs/drever_a.shtml`, accessed March 2009.

[15] ———, "Virus Description: Cardblock.A," `http://www.f-secure.com/v-descs/cardblock_a.shtml`, accessed March 2009.

[16] Schmidt, A.-D., H.-G. Schmidt, L. Batyuk, J. H. Clausen, S. A. Camtepe, and S. Albayrak (2009) "Smartphone Malware Evolution Revisited: Android Next Target?" in *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)*.

[17] Oberheide, J. (2010) "Android Hax," in *Proceedings of SummerCon*.

[18] The H Open Source (2010), "Android holes allow secret installation of apps," `http://www.h-online.com/open/news/item/Android-holes-allow-secret-installation-of-apps-1134940.html`.

[19] First Tech Credit Union (2009), "Security Fraud: Rogue Android Smartphone app created," `http://www.firsttechcu.com/home/security/fraud/security_fraud.html`.

[20] Kasperskey Lab (2010), "First SMS Trojan detected for smartphones running Android," `http://www.kaspersky.com/news?id=207576158`.

[21] Porras, P., H. Saidi, and V. Yegneswaran (2009) *An Analysis of the Ikee.B (Duh) iPhone Botnet*, *Tech. rep.*, SRI International, `http://mtc.sri.com/iPhone/`.

[22] Polychronakis, M., P. Mavrommatis, and N. Provos (2008) "Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware," in *Proceedings of the USENIX Workshop on Large-Scale and Emergent Threats (LEET)*.

[23] Dagon, D., T. Martin, and T. Starner (2004) "Mobile Phones as Computing Devices: The Viruses are Coming!" *IEEE Pervasive Computing*, **3**(4), pp. 11–15.

[24] F-Secure Corporation, "Virus Description: Pbstealer.A," `http://www.f-secure.com/v-descs/pbstealer_a.shtml`, accessed March 2009.

[25] ———, "Virus Description: Viver.A," `http://www.f-secure.com/v-descs/trojan_symbos_viver_a.shtml`, accessed March 2009.

[26] Guo, C., H. J. Wang, and W. Zhu (2004) "Smart-Phone Attacks and Defenses," in *Proceedings of the 3rd Workshop on Hot Topics in Networks (HotNets)*.

[27] Cheng, J., S. H. Wong, H. Yang, and S. Lu (2007) "SmartSiren: Virus Detection and Alert for Smartphones," in *Proceedings of the International conference on Mobile Systems, Applications, and Services (MobiSys)*.

[28] (2010), "Mobile game trojan calls the South Pole," `http://www.gamepron.com/news/2010/05/31/mobile-game-trojan-calls-the-south-pole/`.

[29] Apple, Inc., "Get Ready for iPhone OS 3.0," `http://developer.apple.com/iphone/program/sdk.html`, accessed April 2009.

[30] Fleizach, C., M. Liljenstam, P. Johansson, G. M. Moelker, and A. Méhes (2007) "Can you Infect Me Now? Malware Propagation in Mobile Phone Networks," in *Proceedings of the ACM Workshop On Rapid Malcode (WORM)*, pp. 61–68.

[31] Shin, D., J. Ahn, and C. Shim (2006) "Progressive Multi Gray-Leveling: A Voice Spam Protection Algorithm," *IEEE Network*, **20**(5), pp. 18–24.

[32] Egele, M., C. Kruegel, E. Kirda, and G. Vigna (2011) "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*.

[33] Blumberg, A. J. and P. Eckersley (2009), "On Locational Privacy, and How to Avoid Losing it Forever," Electronic Frontier Foundation, `http://www.eff.org/wp/locational-privacy`.

[34] Lookout (2010), "Update and Clarification of Analysis of Mobile Applications at Blackhat 2010," `http://blog.mylookout.com/2010/07/mobile-application-analysis-blackhat/`.

[35] Apple, Inc. (2011), "Apple's Mac App Store Opens for Business," `http://www.apple.com/pr/library/2011/01/06macappstore.html`.

[36] Barrera, D. and P. C. van Oorschot (2011) "Secure Software Installation on Smartphones," (to appear).

[37] Stone, B. (2009), "Amazon Erases Orwell Books From Kindle," The New York Times, `http://www.nytimes.com/2009/07/18/technology/companies/18amazon.html`.

[38] Ellison, C. and B. Schneier (2000) "Ten Risks of PKI: What You're Note Being Told About Public Key Infrastructure," *Computer Security Journal*, **16**(1), pp. 1–7.

[39] Lennon, M. (2010), "Sophisticated New Android Trojan "Geinimi" Spreading in China," Security Week, `http://www.securityweek.com/sophisticated-new-android-trojan-geinimi-spreading-china`.

[40] Independent Security Evaluators, "Exploiting Android," `http://securityevaluators.com/content/case-studies/android/index.jsp`.

[41] Enck, W., M. Ongtang, and P. McDaniel (2009) "Understanding Android Security," *IEEE Security & Privacy Magazine*, **7**(1), pp. 50–57.

[42] Anderson, J. P. (1972) *Computer Security Technology Planning Study, ESDTR-73-51*, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, (Also available as Vol. I, DITCAD-758206. Vol. II DITCAD-772806).

[43] Saltzer, J. H. (1974) "Protection and the Control of Information Sharing in Multics," *Communications of the ACM*, **17**(7).

[44] ——— (1974) "Ongoing Research and Development on Information Protection," *ACM SIGOPS Operating Systems Review*, **8**(3).

[45] Jaeger, T. (2008) *Operating System Security, Synthesis Lectures on Inforation Security Privacy and Trust.*

[46] Lampson, B. (1971) "Protection," in *Proceedings of the Fifth Princeton Symposium of Information Science and Systems*, pp. 437–443.

[47] Harrison, M., W. Ruzzo, and J. Ullman (1976) "Protection in Operating Systems," *Communications of the ACM*, **19**(8), pp. 461–471.

[48] Fraim, L. J. (1983) "Scomp: A Solution to the Multilevel Security Problem," *IEEE Computer*, **16**(7), pp. 26–24.

[49] Shockley, W. R., T. F. Tao, and M. F. Thompson (1988) "An Overview of the GEMSOS Class A1 Technology and Application Experience," in *Proceedings of the 11th National Computer Security Conference*, pp. 238–245.

[50] Rushby, J. M. (1981) "Design and Verification of Secure Systems," *ACM SIGOPS Operating Systems Review*, **15**(5).

[51] ——— (1982) "Proof of Separability: A Verification Technique for a Class of Security Kernels," in *Proceedings of the 5th International Symposium on Programming.*

[52] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. R. an dAVadis Tevanian, and M. Young (1986) "Mach: A New Kernel Foundation For UNIX Development," in *Proceedings of the Summer USENIX Conference.*

[53] Liedtke, J. (1993) "Improving IPC by Kernel Design," in *Proceedings of the 14th ACM Symposium on Operating Sysetms Principles (SOSP).*

[54] ——— (1995) "On μ-Kernel Construction," in *Proceedings of the 15th ACM Symposium on Operating Sysetms Principles (SOSP).*

[55] Goldberg, R. (1972) *Arcitechural Principles for Virtual Computer Systems*, Ph.D. thesis, Harvard University.

[56] ——— (1974) "Survey of Virtual Machine Research," *IEEE Computer Magazine*, **7**, pp. 34–45.

[57] Creasy, R. J. (1981) "The Origin of the VM/370 Time-Sharing System," *IBM J. Research and Development*, **25**(5), pp. 483–490.

[58] Kelem, N. L. and R. J. Feiertag (1991) "A Sepration Model for Virtual Machine Monitors," in *Proceedings of the IEEE Symposium on Research in Security and Privacy.*

[59] Sugerman, J., G. Venkitachalam, and B.-H. Lim (2001) "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," in *Proceedings of the USENIX Annual Technical Conference.*

[60] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003) "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP).*

[61] Garfinkel, T., B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh (2003) "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 193–206.

[62] Sailer, R., T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doom (2005) "Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pp. 276–285.

[63] Denning, D. E. (1976) "A Lattice Model of Secure Information Flow," *Communications of the ACM*, **19**(5), pp. 236–243.

[64] Bell, D. E. and L. J. LaPadula (1973) *Secure Computer Systems: Mathematical Foundations*, *Tech. Rep. MTR-2547, Vol. 1*, MITRE Corp., Bedford, MA.

[65] Biba, K. J. (1977) *Integrity Considerations for Secure Computer Systems*, *Tech. Rep. MTR-3153*, MITRE.

[66] Clark, D. D. and D. Wilson (1987) "A Comparison of Military and Commercial Security Policies," in *Proceedings IEEE Symposium on Security and Privacy.*

[67] Shankar, U., T. Jaeger, and R. Sailer (2006) “Toward Autmoated Information-Flow Integrity Verification for Security-Critical Applications,” in *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)*.

[68] Li, N., Z. Mao, and H. Chen (2007) “Usable Mandatory Integrity Protection for Operating Systems,” in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 164–178.

[69] Sun, W., R. Sekar, G. Poothia, and T. Karandikar (2008) “Practical Proactive Integrity Protection: A Basis for Malware Defense,” in *Proceedings of the IEEE Symposium on Security and Privacy*.

[70] Vandebogart, S., P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières (2007) “Labels and Event Processes in the Asbestos Operating System,” *ACM Transactions on Computer Systems (TOCS)*, **25**(4).

[71] Zeldovich, N., S. Boyd-Wickizer, E. Kohler, and D. Mazières (2006) “Making Information Flow Explicit in HiStar,” in *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)*, pp. 263–278.

[72] Krohn, M., A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris (2007) “Information Flow Control for Standard OS Abstractions,” in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pp. 321–334.

[73] Myers, A. C. and B. Liskov (2000) “Protecting Privacy Using the Decentralized Label Model,” *ACM Transactions on Software Engineering and Methodology*, **9**(4), pp. 410–442.

[74] Badger, L., D. Sterne, D. Sherman, K. Walker, and S. Haghighat (1995) “A Domain and Type Enforcement UNIX Prototype,” in *Proceedings of the Fifth USENIX UNIX Security Symposium*.

[75] Boebert, W. E. and R. Y. Kain (1985) “A Practical Alternative to Hierarchical Integrity Policies,” in *Proceedings of the 8th National Computer Security Conference*.

[76] National Security Agency, “Security-Enhanced Linux (SELinux),” `http://www.nsa.gov/selinux`.

[77] Spencer, R., S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau (1999) “The Flask Security Architecture: System Support for Diverse Security Policies,” in *Proceedings of the 8th USENIX Security Symposium*, pp. 123–139.

[78] Ferraiolo, D., J. Cugini, and D. R. Kuhn (1995) “Role-Based Access Control (RBAC): Features and Motivations,” in *Proceedings of 11th Annual Computer Security Application Conference (ACSAC)*.

[79] Novell (Accessed January 2011), "AppArmor Application Security for Linux," `http://www.novell.com/linux/security/apparmor/`.

[80] Cowan, C., S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor (2000) "SubDomain: Parsimonious Server Security," in *Proceedings of the 14th USENIX conference on System administration*.

[81] Saltzer, J. and M. Schroeder (1975) "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, **63**(9).

[82] Hardy, N. "The Confused Deputy: (or why capabilities might have been invented)," *SIGOPS Operating Systems Review*, **22**(4), pp. 36–38.

[83] Wulf, W., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack (1974) "HYDRA: The Kernel of a Multiprocessor Operating Systems," *Communications of the ACM*, **17**(6).

[84] Boebert, W. E. (1984) "On the Inability of an Unmodified Capability Machine to Enforce the *-property," in *Proceedings of the DoD/NBS Computer Security Conference*, pp. 291–293.

[85] Karger, P. A. and A. J. Herbert (1984) "An Augmented Capability Architecture to Support Lattice Security and Traceability of Access," in *Proceedings of the IEEE Symposium on Security and Privacy*.

[86] Karger, P. A. (1988) *Improving Security and Performance for Capability Systems*, Ph.D. thesis, University of Cambridge.

[87] Shapiro, J. S. (1999) *EROS: A Capability System*, Ph.D. thesis, University of Pennsylvania.

[88] Goldberg, I., D. Wagner, R. Thomas, and E. Brewer (1996) "A Secure Environment for Untrusted Helper Applicationos: Confining the Wily Hacker," in *Proceedings of the USENIX Security Symposium*.

[89] Jaeger, T., A. Rubin, and A. Prakash (1996) "Building Systems That Flexibly Control Downloaded Executable Content," in *Proceedings of the 6th USENIX UNIX Security Symposium*, pp. 131–148.

[90] Acharya, A. and M. Raje (2000) "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," in *Proceedings of the 9th USENIX Security Symposium*.

[91] Lai, N. and T. Gray (1988) "Strengthening Discresionary Access Controls to Inhibit Trojan Horses and Computer Viruses," in *Proceedings of the 1988 USENIX Summer Symposium*, pp. 275–286.

[92] Berman, A., V. Bourassa, and E. Selberg (1995) "TRON: Process-Specific File Protection for the UNIX Operating System," in *Proceedings of the USENIX Technical Conference*, pp. 165–175.

[93] Seaborn, M., "Plash," `http://plash.beasts.org`.

[94] Stiegler, M., A. Karp, K.-P. Yee, and M. Miller (2004) *Polaris: Virus Safe Computing for Windows XP*, *Tech. Rep. HPL-2004-221*, HP Laboratories Palo Alto.

[95] Wichers, D., D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo (1990) "PACL's: An Access Control List Approach to Anti-viral Security," in *Proceedings of the 13th National Computer Security Conference*, pp. 340–349.

[96] Enck, W., P. McDaniel, and T. Jaeger (2008) "PinUP: Pinning User Files to Known Applications," in *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*.

[97] Enck, W., S. Rueda, Y. Sreenivasan, J. Schiffman, L. S. Clair, T. Jaeger, and P. McDaniel (2007) "Protecting Users from "Themselves"," in *Proceedings of the 1st ACM Computer Security Architectures Workshop*.

[98] Ioannidis, S., S. Bellovin, and J. Smith (2002) "Sub-Operating Systems: A New Approach to Application Security," in *Proceedings of ACM SIGOPS European workshop*, pp. 108–115.

[99] Snowberger, P. and D. Thain (2005) *Sub-Identities: Towards Operating System Support for Distributed System Security*, *Tech. Rep. 2005-18*, University of Notre Dame, Department of Computer Science and Engineering.

[100] Schmid, M., F. Hill, and A. Gosh (2002) "Protecting Data from Malicious Software," in *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*.

[101] Sailer, R., X. Zhang, T. Jaeger, and L. van Doom (2004) "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *Proceedings of the 13th USENIX Security Symposium*.

[102] Trusted Computing Group (2007), "TCG mobile reference architecture specification version 1.0," `https://www.trustedcomputinggroup.org/specs/mobilephone/tcg-mobile-reference-architecture-1.0.pdf`.

[103] Zhang, X., O. Aciiçmez, and J.-P. Seifert (2007) "A Trusted Mobile Phone Reference Architecture via Secure Kernel," in *Proceedings of the ACM workshop on Scalable Trusted Computing*, pp. 7–14.

[104] ——— (2009) "Building Efficient Integrity Measurement and Attestation for Mobile Phone Platforms," in *Proceedings of the First International ICST Conference on Security and Privacy in Mobile Information and Communication Systems (MobiSec)*.

[105] Nauman, M., S. Khan, X. Zhang, and J.-P. Seifert (2010) "Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform," in *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*.

[106] Muthukumaran, D., A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger (2008) "Measuring Integrity on Mobile Phone Systems," in *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 155–164.

[107] Shabtai, A., Y. Fledel, and Y. Elovici (2010) "Securing Android-Powered Mobile Devices Using SELinux," *IEEE Security and Privacy Magazine*.

[108] VMware, Inc., "VMware Mobile Virtualization Platform," `http://www.vmware.com/products/mobile/`, accessed January 2011.

[109] Open Kernel Labs, "OK:Android," `http://www.ok-labs.com/products/ok-android`, accessed January 2011.

[110] Lee, S.-M., S. bum Suh, B. Jeong, and S. Mo (2008) "A Multi-Layer Mandatory Access Control Mechanism for Mobile Devices Based on Virtualization," in *Proceedings of the 5th IEEE Consumer Communications and Networking Conference (CCNC)*.

[111] Winter, J. (2008) "Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms," in *Proceedings of the 3rd ACM workshop on Scalable Trusted Computing (STC)*.

[112] Mulliner, C., G. Vigna, D. Dagon, and W. Lee (2006) "Using Labeling to Prevent Cross-Service Attacks Against Smart Phones," in *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[113] Ion, I., B. Dragovic, and B. Crispo (2007) "Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[114] Desmet, L., W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe (2008) "Security-by-contract on the .NET platform," *Information Security Technical Report*, **13**(1), pp. 25–32.

[115] Nauman, M., S. Khan, and X. Zhang (2010) "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *Proceedings of ASIACCS*.

[116] Conti, M., V. T. N. Nguyen, and B. Crispo (2010) "CRePE: Context-Related Policy Enforcement for Android," in *Proceedings of the 13th Information Security Conference (ISC)*.

[117] Ongtang, M., S. McLaughlin, W. Enck, and P. McDaniel (2009) "Semantically Rich Application-Centric Security in Android," in *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, pp. 340–349.

[118] Ongtang, M., K. Butler, and P. McDaniel (2010) "Porscha: Policy Oriented Secure Content Handling in Android," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*.

[119] Karlson, A. K., A. B. Brush, and S. Schechter (2009) "Can I Borrow Your Phone? Understanding Concerns When Sharing Mobile Phones," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*.

[120] Liu, Y., A. Rahmati, Y. Huang, H. Jang, L. Zhong, Y. Zhang, and S. Zhang (2009) "xShare: Supporting Impromptu Sharing of Mobile Phones," in *Proceedings of the International conference on Mobile Systems, Applications, and Services (MobiSys)*.

[121] Ni, X., Z. Yang, X. Bai, A. C. Champion, and D. Xuan (2009) "DiffUser: Differentiated User Access Control on Smartphones," in *Proceedings of the 5th IEEE Workshop on Wireless and Sensor Networks Security (WSNS)*.

[122] Shin, W., S. Kiyomoto, K. Fukushima, and T. Tanaka (2010) "A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework," in *Proceedings of the International Conference on Social Computing*.

[123] Shin, W., S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka (2010) "A Small but Non-negligible Flaw in the Android Permission Scheme," in *Proceedings of the International Symposium on Policies for Distributed Systems and Networks*.

[124] Chaudhuri, A. (2009) "Language-Based Security on Android," in *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*.

[125] Fuchs, A. P., A. Chaudhuri, and J. S. Foster, "ScanDroid: Automated Security Certification of Android Applications," `http://www.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf`, accessed January 11, 2011.

[126] Venugopal, D. and G. Hu (2008) "Efficient Signature Based Malware Detection on Mobile Devices," *Mobile Information Systems*, **4**(1).

[127] Bose, A., X. Hu, K. G. Shin, and T. Park (2008) "Behavioral Detection of Malware on Mobile Handsets," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[128] Shabtai, A., U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss (2011) ""Andromaly": A Behavioral Malware Detection Framework for Android Devices," *Journal of Intelligent Information Systems*, published online January 2011.

[129] Nash, D. C., T. L. Martin, D. S. Ha, and M. S. Hsiao (2005) "Towards an Intrusion Detection System for Battery Exhaustion Attacks on Mobile Computing Devices," in *Proceedings of the 3rd International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*.

[130] Kim, H., J. Smith, and K. G. Shin (2008) "Detecting Energy-Greedy Anomalies and Mobile Malware Variants," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 239–252.

[131] Miettinen, M., P. Halonen, and K. Hatonen (2006) "Host-Based Intrusion Detection for Advanced Mobile Devices," in *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA).*

[132] Schmidt, A.-D., F. Peters, F. Lamour, C. Scheel, S. A. Camtepe, and S. Albayrak (2009) "Monitoring Smartphones for Anomaly Detection," *Mobile Networks and Applications*, **14**(1), pp. 92–106.

[133] Chun, B.-G. and P. Maniatis (2009) "Augmented Smartphone Applications Through Clone Cloud Execution," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems (HotOS).*

[134] Portokalidis, G., P. Homburg, K. Anagnostakis, and H. Bos (2010) "Paranoid Android: Versatile Protection For Smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC).*

[135] Oberheide, J., K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian (2008) "Virtualized In-Cloud Security Services for Mobile Devices," in *Proceedings of the 1st Workshop on Virtualization in Mobile Computing.*

[136] Oberheide, J., E. Cooke, and F. Jahanian (2008) "CloudAV: N-Version Antivirus in the Network Cloud," in *Proceedings of the 17th USENIX Security Symposium.*

[137] Migliavacca, M., I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch (2010) "DEFCon: High-Performance Event Processing with Information Security," in *Proceedings of the USENIX Annual Technical Conference.*

[138] Wang, X., Z. Li, N. Li, and J. Y. Choi (2008) "PRECIP: Towards Practical and Retrofittable Confidential Information Protection," in *Proceedings of 15th Network and Distributed System Security Symposium (NDSS08).*

[139] Sabelfeld, A. and A. C. Myers (2003) "Language-based information-flow security," *IEEE Journal on Selected Areas in Communication*, **21**(1), pp. 5–19.

[140] Myers, A. C. (1999) "JFlow: Practical Mostly-Static Information Flow Control," in *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL).*

[141] Heintze, N. and J. G. Riecke (1998) "The SLam Calculus: Programming with Secrecy and Integrity," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pp. 365–377.

[142] Roy, I., D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel (2009) "Laminar: Practical Fine-Grained Decentralized Information Flow Control," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pp. 63–74.

[143] Hicks, B., K. Ahmadizadeh, and P. McDaniel (2006) "Understanding Practical Application Development in Security-Typed Languages," in *22st Annual Computer Security Applications Conference (ACSAC)*, Miami, Fl, pp. 153–164.

[144] Denning, D. E. and P. J. Denning (1977) "Certification of Programs for Secure Information Flow," *Communications of the ACM*, **20**(7).

[145] Newsome, J. and D. Song (2005) "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*.

[146] Qin, F., C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu (2006) "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society Washington, DC, USA, pp. 135–148.

[147] Clause, J., W. Li, and A. Orso (2007) "Dytan: A Generic Dynamic Taint Analysis Framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ACM New York, NY, USA, pp. 196–206.

[148] Yin, H., D. Song, M. Egele, C. Kruegel, and E. Kirda (2007) "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *Proceedings of the 14th ACM conference on Computer and Communications Security*, pp. 116–127.

[149] Egele, M., C. Kruegel, E. Kirda, H. Yin, and D. Song (2007) "Dyanmic Spyware Analysis," in *Proceedings of the USENIX Annual Technical Conference*, pp. 233–246.

[150] Zhu, D., J. Jung, D. Sung, T. Kohno, and D. Wetherall (2009) *Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks*, *Tech. Rep. EECS-2009-145*, Department of Computer Science, UC Berkeley.

[151] Costa, M., J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham (2005) "Vigilante: End-to-End Containment of Internet Worms," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pp. 133–147.

[152] Vachharajani, N., M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August (2004) "RIFLE: An Architectural Framework for User-Centric Information-Flow Security," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society Washington, DC, USA, pp. 243–254.

[153] Crandall, J. R. and F. T. Chong (2004) "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *Proceedings of the International Symposium on Microarchitecture*, pp. 221–232.

[154] Suh, G. E., J. W. Lee, D. Zhang, and S. Devadas (2004) "Secure Program Execution via Dynamic Information Flow Tracking," in *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 85–96.

[155] Chow, J., B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum (2004) "Understanding Data Lifetime via Whole System Simulation," in *Proceedings of the 13th USENIX Security Symposium.*

[156] Cheng, W., Q. Zhao, B. Yu, and S. Hiroshige (2006) "TaintTrace: Efficient Flow Tracing with Dyanmic Binary Rewriting," in *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, pp. 749–754.

[157] Ho, A., M. Fetterman, C. Clark, A. Warfield, and S. Hand (2006) "Practical Taint-Based Protection using Demand Emulation," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, pp. 29–41.

[158] Xu, W., S. Bhatkar, and R. Sekar (2006) "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," in *Proceedings of the USENIX Security Symposium*, pp. 121–136.

[159] Lam, L. C. and T. cker Chiueh (2006) "A General Dynamic Information Flow Tracking Framework for Security Applications," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pp. 463–472.

[160] Saxena, P., R. Sekar, and V. Puranik (2008) "Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking," in *Proceedings of the IEEE/ACM symposium on Code Generation and Optimization (CGO)*, pp. 74–83.

[161] Haldar, V., D. Chandra, and M. Franz (2005) "Dynamic Taint Propagation for Java," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pp. 303–311.

[162] Halfond, W. G., A. Orso, and P. Manolios (2008) "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," *IEEE Transactions on Software Engineering*, **34**(1), pp. 65–81.

[163] Chandra, D. and M. Franz (2007) "Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC).*

[164] Nair, S. K., P. N. Simpson, B. Crispo, and A. S. Tanenbaum (2007) *Design and Implementation of a Virtual Machine Based Information Flow Control System, Tech. Rep. IR-CS-040*, Department of Computer Science, Vrije Universiteit.

[165] Vogt, P., F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna (2007) "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *Proceedings of the 14th Network & Distributed System Security Symposium.*

[166] Yip, A., X. Wang, N. Zeldovich, and M. F. Kaashoek (2009) "Improving Application Security with Data Flow Assertions," in *Proceedings of the ACM Symposium on Operating Systems Principles.*

[167] Ashcraft, K. and D. Engler (2002) "Using Programmer-Written Compiler Extensions to Catch Security Holes," in *Proceedings of the IEEE Symposium on Security and Privacy.*

[168] Chen, H., D. Dean, and D. Wagner (2004) "Model Checking One Million Lines of C Code," in *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS).*

[169] Schwarz, B., H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu (2005) "Model Checking an Entire Linux Distribution for Security Violations," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC).*

[170] Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner (2006) "Thorough Static Analysis of Device Drivers," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys).*

[171] Ball, T. and S. K. Rajamani (2001) "Automatically Validating Temporal Safety Properties of Interfaces," in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software.*

[172] Ware, M. S. and C. J. Fox (2008) "Securing Java Code: Heuristics and an Evaluation of Static Analysis Tools," in *Proceedings of the Workshop on Static Analysis (SAW).*

[173] Hovemeyer, D. and W. Pugh (2004) "Finding Bugs is Easy," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*

[174] Livshits, V. B. and M. S. Lam (2005) "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *Proceedings of the 14th USENIX Security Symposium.*

[175] Felmetsger, V., L. Cavedon, C. Kruegel, and G. Vigna (2010) "Toward Automated Detection of Logic Vulnerabilities in Web Applications," in *Proceedings of the 19th USENIX Security Symposium.*

[176] Jovanovic, N., C. Kruegel, and E. Kirda (2010) "Static Analysis for Detection Taint-style Vulnerabilities in Web Applications," *Journal of Computer Security*, **18**(5).

[177] Balzarotti, D., M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna (2008) "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," in *Proceedings of the IEEE Symposium on Security and Privacy.*

[178] Kirda, E., C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer (2006) "Behavior-based Spyware Detection," in *Proceedings of the 15th USENIX Security Symposium*, pp. 273–288.

[179] Jung, J., A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno (2008) "Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing," in *Proceedings of the 15th ACM conference on Computer and Communications Security*, ACM New York, NY, USA, pp. 279–288.

[180] Yumerefendi, A. R., B. Mickle, and L. P. Cox (2007) "TightLip: Keeping Applications from Spilling the Beans," in *Proceedings of the 4th USENIX Symposium on Network Systems Design & Implementation (NSDI)*, pp. 159–172.

[181] F-Secure Corporation (2007), "Just because it's Signed doesn't mean it isn't spying on you," `http://www.f-secure.com/weblog/archives/00001190.html`.

[182] ———, "Virus Description: Worm:SymbOS/Yxe.A," `http://www.f-secure.com/v-descs/worm_symbos_yxe.shtml`, accessed March 2009.

[183] Cranor, L. F. (2003) "P3P: Making Privacy Policies More Useful," *IEEE Security & Privacy magazine*, **1**(6), pp. 50–55.

[184] (2005) *ISO/IEC 15408-1 Information technology - Security techniques - Evaluation criteria for IT security - Part 1: Introduction and general model.*

[185] Haley, C. B., J. D. Moffett, R. Laney, and B. Nuseibeh (2006) "A framework for security requirements engineering," in *SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems*, pp. 35–42.

[186] Chen, P., M. Dean, D. Ojoko-Adams, H. Osman, L. Lopez, and N. Xie (2004) *System Quality Requirements Engineering (SQUARE) Methodology: Case Study on Asset Management System, Tech. Rep. CMU/SEI-2004-SR-015*, Software Engineering Institute, Carnegie Mellon University.

[187] Mead, N. R. (2007) *How To Compare the Security Quality Requirements Engineering (SQUARE) Method with Other Methods, Tech. Rep. CMU/SEI-2007-TN-021*, Software Engineering Institute, Carnegie Mellon University.

[188] Mellado, D., E. Fernández-Medina, and M. Piattini (2006) "Applying a Security Requirements Engineering Process," *LNCS: Computer Security – ESORICS 2006*, **4189/2006**, pp. 192–206.

[189] ——— (2007) "A common criteria based security requirements engineering process for the development of secure information systems," *Computer Standards & Interfaces*, **29**(2), pp. 244–253.

[190] Open Web Application Security Project (OWASP) Foundation (2009), "CLASP (Comprehensive, Lightweight Application Security Process) Project," `http://www.owasp.org/index.php/OWASP_CLASP_Project`.

[191] McDermott, J. and C. Fox (1999) "Using Abuse Case Models for Security Requirements Analysis," in *Proceedings of the 15th Annual Computer Security Applications Conference*.

[192] Sindre, G. and A. L. Opdahl (2004) "Eliciting security requirements with misuse cases," *LNCS: Requirements Engineering*, pp. 34–44.

[193] Hatebur, D., M. Heisel, and H. Schmidt (2007) "A Pattern System for Security Requirements Engineering," in *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*, pp. 356–365.

[194] Schumacher, M., E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad (2009) *Security Patterns: Integrating Security and Systems Engineering*, Wiley.

[195] Yoshioka, N., H. Washizaki, and K. Maruyama (2008) "A survey on security patterns," *Progress in Informatics, Special issue: The future of software engineering for security and privacy*, **5**, pp. 35–47.

[196] Lipner, S. and M. Howard (2005), "The Trustworthy Computing Security Development Lifecycle," `http://msdn.microsoft.com/en-us/library/ms995349.aspx`.

[197] Oracle (2009), "Oracle Software Security Assurance," `http://www.oracle.com/security/software-security-assurance.html`.

[198] Moffett, J. D., C. B. Haley, and B. Nuseibeh (2004) *Core Security Requirements Artefacts*, *Tech. rep.*, Open University, UK.

[199] Rosenblatt, J. (2011), "Apple Sued Over Applications Giving Information to Advertisers," Bloomberg Businessweek, `http://www.businessweek.com/news/2011-01-05/apple-sued-over-applications-giving-information-to-advertisers.html`.

[200] Lookout (2010), "Introducing the App Genome Project," `http://blog.mylookout.com/2010/07/introducing-the-app-genome-project/`.

[201] "WhatApp," `http://www.whatapp.org`, accessed April 2010.

[202] Newsome, J., S. McCamant, and D. Song (2009) "Measuring Channel Capacity to Distinguish Undue Influence," in *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*.

[203] Schwartz, E. J., T. Avgerinos, and D. Brumley (2010) "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*.

[204] Slowinska, A. and H. Bos (2009) "Pointless Tainting? Evaluating the Practicality of Pointer Tainting," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, pp. 61–74.

[205] Cox, L. P. and P. Gilbert (2009) *RedFlag: Reducing Inadvertent Leaks by Personal Machines*, *Tech. Rep. TR-2009-02*, Duke University.

[206] "Android Market," `http://market.android.com`.

[207] "Android," `http://www.android.com`.

[208] "Apache Harmony – Open Source Java Platform," `http://harmony.apache.org`.

[209] Liang, S. (1999) *Java Native Interface: Programmer's Guide and Specification*, Prentice Hall PTR.

[210] King, D., B. Hicks, M. Hicks, and T. Jaeger (2008) "Implicit Flows: Can't Live with 'Em, Can't Live without 'Em," in *Proceedings of the International Conference on Information Systems Security.*

[211] "Google Maps for Mobile," `http://www.google.com/mobile/products/maps.html`.

[212] "Flurry Mobile Application Analytics," `http://www.flurry.com/product/technical-info.html`.

[213] Pendragon Software Corporation, "CaffeineMark 3.0," `http://www.benchmarkhq.ru/cm30/`.

[214] Raphel, J. (2010) "Google: Android wallpaper apps were not security threats," *Computerworld.*

[215] Goodin, D. (2009), "Backdoor in Top iPhone Games Stole User data, Suit Claims," The Register, `http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit/`.

[216] BBC News (2009), "New iPhone worm can act like botnet say experts," `http://news.bbc.co.uk/2/hi/technology/8373739.stm`.

[217] Kralevich, N. (2010), "Best Practices for Handling Android User Data," `http://android-developers.blogspot.com/2010/08/best-practices-for-handling-android.html`.

[218] Vallee-Rai, R., E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan (2000) "Optimizing Java Bytecode using the Soot Framework: Is it Feasible?" in *International Conference on Compiler Construction, LNCS 1781*, pp. 18–34.

[219] Gagnon, E., L. J. Hendren, and G. Marceau (2000) "Efficient Inference of Static Types for Java Bytecode," in *SAS '00: Proc. of the 7th International Symposium on Static Analysis*, Springer-Verlag, pp. 199–219.

[220] Bellamy, B., P. Avgustinov, O. de Moor, and D. Sereni (2008) "Efficient Local Type Inference," in *OOPSLA'07: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

[221] Mycroft, A. (1999) "Type-Based Decompilation (or Program Reconstruction via Type Reconstruction)," in *Proc. of the 8th European Symposium on Programming Languages and Systems*, Springer-Verlag, pp. 208–223.

[222] "JD Java Decompiler," `http://java.decompiler.free.fr/`.

[223] "Jad - the fast JAva Decompiler," `http://www.kpdus.com/jad.html`.

[224] "Fernflower - Java Decompiler," `http://www.reversed-java.com/fernflower/`.

[225] "Fortify 360 Source Code Analyzer (SCA)," `https://www.fortify.com/products/fortify360/source-code-analyzer.html`.

[226] Schlegel, R., K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang (2011) "Soundminer: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *Proceedings of the ISOC Annual Network and Distributed System Security Symposium (NDSS)*.

[227] Burns, J. (2008), "Developing Secure Mobile Applications for Android," iSEC Partners, `http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf`.

[228] AdMob, "AdMob Android SDK: Installation Instructions," `http://www.admob.com/docs/AdMob_Android_SDK_Instructions.pdf`, accessed November 2010.

[229] Spencer E. Ante (2010), "Banks Rush to Fix Security Flaws in Wireless Apps," `http://online.wsj.com/article/SB10001424052748703805704575594581203248658.html`.

[230] Johns, T. (2010), "Securing Android LVL Applications," `http://android-developers.blogspot.com/2010/09/securing-android-lvl-applications.html`.

[231] Storm, D. (2010) "Zombies and Angry Birds attack: mobile phone malware," *Computerworld*.

[232] Android Security Discussions Mailing List (2010), "Third party firewall/antivirus," `http://groups.google.com/group/android-security-discuss/browse_thread/thread/d3154b8a65003366/`.

[233] Pruett, C. (2011), "Gingerbread NDK Awesomeness," `http://android-developers.blogspot.com/2011/01/gingerbread-ndk-awesomeness.html`.

[234] Ziegler, C. (2010), "Motoroal reponds to Droid X bootloader controversy, says eFuse isn't there to break the phone," Engadget, `http://www.engadget.com/2010/07/16/motorola-responds-to-droid-x-bootloader-controversy-says-efuse/`.

[235] Kamara, S. and K. Lauter (2010) "Cryptographic Cloud Storage," in *Proceedings of the Financial Cryptography: Workshop on Real-Life Cryptographic Protocols and Standardization*.

Vita

William Harold Enck

Education

The Pennsylvania State University, University Park, PA. Ph.D in Computer Science and Engineering, Spring 2011.
The Pennsylvania State University, University Park, PA. M.S. in Computer Science and Engineering, Spring 2006.
The Pennsylvania State University, University Park, PA. B.S. with Honors and Highest Distinction in Computer Engineering, Spring 2004.

Awards and Honors

- Pennsylvania State University Alumni Association Dissertation Award (2011)
- Penn State CSE Graduate Research Assistant Award (2010)
- Best Paper, 25th Annual Computer Security Applications Conference (2009)
- NSF Graduate Research Fellowship, Honorable Mention (2006)
- H. Thomas and Dorothy Willits Hallowell Scholarship, Penn State University (2003)
- Chris Mader Scholarship, Penn State University (2002)
- Lockheed Martin Engineering Scholars Award, Penn State University (2002)
- Richard A. McQuade Memorial Scholarship, Penn State University (2001)

Professional Experience

- **Research Intern** *Intel Labs*, Seattle, WA, Summer 2009
  Designed, developed, and implemented the TaintDroid tracking system for realtime privacy analysis of smartphone applications.
- **co-Instructor** *Penn State University*, State College, PA, Spring 2009
  co-Instructor for CSE597a, a graduate seminar on mobile phone operating systems security.
- **Instructor** *Penn State University*, State College, PA, Summer 2007
  Instructor for EE/CSE458, the senior undergraduate course on computer networking.
- **Research Intern** *AT&T Research*, Florham Park, NJ, Summer 2006
  Developed a system for massive-scale automated router configuration deployment.
- **Research Assistant** *Penn State University*, State College, PA, 2005 - 2011
  Led research projects in systems security, including mobile operating systems, desktop operating systems, hardware security, and secure network protocols.
- **Teaching Assistant** *Penn State University*, State College, PA, Spring 2005
  Provided hands-on aid for CSE473, a laboratory based undergraduate course on embedded systems development.
- **Summer Intern** *IBM Corp.*, Poughkeepsie, NY, Summer 2003.
  Performed error propagation logic testing for the zSeries mainframe processor hardware caches.
- **Systems Administrator** *Lebanon MobileFone Inc.*, Lebanon, PA, Summers 2000 - 2002 and 2004.
  Administrated Web and Email servers. Designed spam detection system for the ISP. Performed corporate and residential installation of wired and wireless broadband equipment. Repaired personal computers and performed phone and on-site customer technical support.