Safe Software Dissemination in Distributed Application Marketplaces

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Timothy Vidas

B.S., Computer Science
M.S., Computer Science

Carnegie Mellon University
Pittsburgh, PA

Jan 2016

Acknowledgments

While dissertations are generally regarded as individual accomplishment, this one, (and I suspect most), would not be possible without the help and support from collaborators, colleagues, friends and family.

In my short time on this planet, I have been fortunate enough to cross paths with many admirable individuals. With some of these people, I had extended opportunity in assuming the role of protégé. While there is not enough room here for me to fully express my gratitude, I would like to take the time to single out several for particular turning points in my life: Robert Fulkerson, for opening my eyes to a new field of study. Dr. Blaine Burnham, for teaching me to think in adversarial ways and offering the first real push toward graduate studies. Alex Nicoll, for years of insightful conversation and collaboration. Chris Eagle, for immeasurable technical mentorship and the opportunity to attend Sk3wl for higher education. Holt Sorenson, for happening to be around for many seminal moments. Brian Caswell, for perpetually reminding me "because, TCP." All are leaders in their own way; all leading by example.

I must express thanks to my fellow researchers and collaborators. Among them, the Carnegie Mellon University "passwords group" and the "Cyber Crime Research group." Beyond the collaborative publications, this set of researchers facilitated rich cognitive discussion and a quick sounding board for new ideas.

I would especially like to thank my thesis committee: Dr. Nicolas Christin (Chair), Dr. Lujo Bauer, Dr. Jason Hong, Dr. Patrick Tague, and Dr. Patrick McDaniel. I place great value on my tenure at Carnegie Mellon, due in no small part to the community of amazing researchers and engineers at CyLab. In particular, I would like to specifically thank Dr. Lorrie Cranor and Dr. Adrian Perrig for mentorship beyond that provided by my committee. The individual, let alone aggregate, teaching and research abilities represented by these handful of professors is humbling. I aspire to one day be looked upon by others as I look upon them.

I carry a deep sense of gratitude toward Dr. Nicolas Christin. Nicolas has dedicated an immense amount of time and energy toward me and my studies. Foremost, I must voice appreciation for the the academic and professional guidance he has provided. However, his pairing of such mentorship with the ability and opportunity for me to individually grow, is understated, and, is not lost upon me. Nicolas took a chance on a part-time applicant and cultivated that into real, worthwhile academic pursuit.

I'd like to thank my father for creating me computer-generated mazes as a child, when computers were only supposed to be used for Lotus 1-2-3. I'd like to thank my mother who always believed in my potential for success. Further, I'd like to thank my parents, for purchasing my first computer instead of sending me to space camp.

I would also like to thank my wife for her immense support and patience. As with many that have trod the path of Ph.D. education, the past several years were littered with various compromises and trade-offs in favor of research. This achievement would otherwise have certainly been unattainable without her unwavering encouragement. Sheila, you are my best friend, my inspiration, and my favorite.

Ph.D. studies are often regarded as life-changing. However, for me, mid-study, life also changed in a very observable way. The desire to improve the world for future generations is nearly a fundamental tenant of basic scientific research, viewing the world through an altruistic and comprehensive lens. Progeny adds substantial clarity to this lens. For this reason, and many more, I am tremendously thankful for the family that has been bestowed upon me.

In a more tangible way, I would like to thank VirusTotal and AV-test for providing access to data and APIs that were used as part of this research. Generally, such enablement facilitates broad research that might otherwise be out of reach. Please continue to provide future scholars with the resources they need to further scientific research.

This thesis is based on work funded in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and DAAD19-02-1-0389 from the Army Research Office, National Science Foundation ITR award CCF-0424422 (Team for Research in Ubiquitous Secure Technology), National Science Foundation IGERT award DGE-0903659 (Usable Privacy and Security) and hardware donation from Google Inc. Any opinions, findings or recommendations made or implied in this work are those of the author(s) and do not necessarily reflect the views of those of the National Science Foundation, Army Research Office, or Google Inc.

Abstract

Today's smartphone represents not only a complex device akin to an always-connected Personal Computer (PC), but also a relatively new mechanism for software dissemination. Unlike the purchase of physical media in brick-and-mortar stores popular since the advent of the PC, modern smartphones favor online software marketplaces that deliver software digitally. The facility for consumers to augment the base functionality of a smartphone has not only acted as a catalyst for the rapid adoption of the smartphone but continues to encourage regular use of the device and marketplace. Concomitant with this consumer adoption, is the new-found attention that mobile platforms receive from miscreants looking to take advantage of the prevalence of smartphones in society.

This dissertation explores the question of how one can provide safety to users of software marketplaces. To this end, we first investigate the notion of mobile-oriented malicious software, both via measurements and experiments anticipating future evolution of the threats. From our measurements, we glean two clear observations. First, the majority of malicious software we measured starts as legitimate software that was subsequently modified to include malicious components. Second, the majority of this software is delivered through a quite distributed set of online software marketplaces.

With an explicit assessment of this malicious software problem, we then turn to discrete mechanisms to provide safety in software marketplaces. We focus on entities with clear equities in the software market systems, namely, software developers, market proprietors and end users.

Smartphone users are regularly required to make security-related decisions informed only with confusing, abstract lists of resources requested by an application. Worse, these lists are often gratuitously over-populated, exacerbating user confusion and ultimately indifference. We endeavor to aid developers in creating safer software by investigating and addressing a specific class of insecure software, those violating the principle of least privilege.

Developers present risk to the end user by unknowingly introducing flaws. Conversely, miscreants knowingly attempt to take advantage of end users. Both developers and miscreants compete to reach users. In this way, application marketplaces are positioned between end users and those creating software. Marketplace proprietors may elect to police their offerings in a bid to make their market safer for end users. Likewise, miscreants seek to evade detection in order to further their nefarious goals. To aid market proprietors, we assessed and expanded upon current techniques of detection evasions. We then designed an evasion-resistant system for mobile malware analysis. Market proprietors may choose to employ a system such as the one we designed. However, policies and procedures relating to malware will always vary among marketplaces, and some will certainly remain seedy.

The main contribution presented in this work is AppIntegrity, a protocol designed to bind application developer to associated domains. AppIntegrity helps ensure that the software an end user is employing is that which the application developer intended, providing not only immediate security value, but also a strong foundation from which other security-related constructs may be built. AppIntegrity links software creators to end users, transcending security risks presented by individual marketplaces. While based on technical underpinnings, AppIntegrity—in most expected implementations—will also include considerable user interaction. For this reason, we not only investigate the technical efficacy of AppIntegrity, but we also investigate user understanding and find promising results.

Contents

List of Figures

List of Tables

List of Acronyms

ADB Android Debug Bridge

AES Advanced Encryption Standard

AGM Arithmetic-Geometric Mean

AOSP Android Open Source Project

API Application Programming Interface

APK Android Application Package

ASLR Address Space Layout Randomization

AST Abstract Syntax Tree

AV Anti-Virus

BIOS Basic Input/Output System

BSD Berkeley Software Distribution

C2 Command and Control

CA Certificate Authority

CD Compact Disc

CDF Cumulative Distribution Function

CERT Computer Emergency Response Team

CFG Control-Flow Graph

CPU Central Processing Unit

CSS Cascading Style Sheet

DARPA Defense Advanced Research Projects Agency

DER Distinguished Encoding Rules

DES Data Encryption Standard

DEX Dalvik Executable Format

DGA Domain name Generator Algorithm

DHCP Dynamic Host Configuration Protocol

DKIM Domain Key Identified Mail

DNS Domain Name System

DRM Digital Rights Management

EULA End User License Agreement

FFT Fast Fourier Transform

FPS Frames Per Second

GCM Google Cloud Messaging

GCS Google Consumer Surveys

GPS Global Positioning System

GPU Graphics Processing Unit

GUI Graphical User Interface

HCI Human-Computer Interaction

HTTP Hypertext Transfer Protocol

ID Identifier

IDE Integrated Development Environment

IDS Intrusion Detection System

IMSI International Mobile Subscriber Identity

IP Internet Protocol

IPC Inter-Process Communication

IR Intermediate Representation

JNI Java Native Interface

JTAG Joint Test Action Group

MAC Media Access Control

MCC Mobile Country Code

MDM Mobile Device Management

MitM Man-in-the-Middle

MMS Multimedia Messaging Service

MNC Mobile Network Code

MTD Memory Technology Device

NDK Native Development Kit

NFC Near-Field Communication

NHTSA National Highway Traffic Safety Administration

NIO New Input/Output

OEM Original Equipment Manufacturer

OS Operating System

OTA Over-the-Air

P2P Peer-to-Peer

PC Personal Computer

PIN Personal Identification Number

PKCS Public Key Crypto-System

PKI Public Key Infrastructure

PXE	Preboot Execution Environment
RAM	Random Access Memory
RF	Radio Frequency
RFS	Robust File System
RMSE	Root-Mean-Square Error
ROM	Read Only Memory
RSA	Rivest-Shamir-Adleman
SD	Secure Digital
SDK	Software Development Kit
SHA1	Secure Hash Algorithm 1
SIM	Subscriber Identity Module
SMS	Short Message Service
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TPM	Trusted Platform Module
UDP	User Datagram Protocol
UI	User Interface
UID	User Identifier
URL	Uniform Resource Locator
USB	Universal Serial Bus
USD	United States Dollar
VM	Virtual Machine
VPN	Virtual Private Network

XML	Extensible Markup Language
XOR	Exclusive OR

Chapter 1

Introduction

The characterization of malware as *malicious* software stems from *intent*. For this reason, the term malware covers a very broad and diverse set of software created with the intent of causing some form of harm. Generalized software designed to con novice users of the most common operating systems is malware, as is specialized software designed solely to perform one detrimental action on a rare industrial control system.

In addition to the maturity gained in common PCs, malware now afflicts contemporary mobile devices. In fact, malware is increasingly focusing on mobile devices, as we will show in section 3.4.1, presenting a growing addition to an already significant problem. This growth is especially concerning given that users and enterprises have relatively little administrative control of today's mobile devices.

When considering contemporary mobile devices, the product space is quite diverse. We draw the line between contemporary and previous generations of "smart devices" with the advent of iOS. Therefore, today, "smart devices" refer to Android and iOS, or those with similar features. Form factor also leads to ambiguity as "smart devices" may refer to smartphones, tablets (with cellular capability or not), and the size in-between: phablets. In some contexts, yet more devices creep in with smartwatches that exhibit similar features but may require another nearby smart device to fully function. Throughout this document, we often bundle all of these groups under the general term "smartphone." When the distinction is important, we more precisely note the subset of smart device.

By many metrics, today's mobile devices can be thought of as kin to general PCs. Processing power and general computing models are similar, and consumers can find familiar statistics about size of Random Access Memory (RAM), number of processing cores, or screen resolution. Technical interfaces are also familiar to consumers, devices employing connectivity like WIFI, Universal Serial Bus (USB), and Bluetooth.

However, smartphones bear little resemblance to general PCs when it comes to administrative capabilities. The base functionality of mobile devices can be extended with mobile applications, but the core abilities to introduce low-level software is limited to enforcing a few administrative policies scoped by the mobile operating systems. This lack of administrative capability obviates the ability to deploy now-common security controls such as firewalls, Intrusion Detection Systems (IDSs), or Anti-Virus (AV). Devices themselves are not general-purpose, and may only be utilized for a few years before becoming antiquated. Dated devices may also be discarded due to any host of reasons including declining performance of a built-in battery, the end of a cellular contract, or lack of software updates.

In other ways, today's mobile devices are at odds with the PC. For instance, consumers are increasingly using mobile devices to perform tasks traditionally done on a PC. Some predictions assert that mobile devices are already supplanting PCs in the home, relegating the PC to that status of "enterprise tool." Apple's CEO, Tim Cook, was recently quoted: "I think if you're looking at a PC, why would you buy a PC anymore? No really, why would you buy one?" Cook further claimed that today's mobile devices are considered "a replacement for a notebook or a desktop for many, many people. They will start using it and conclude they no longer need to use anything else, other than their phones" [123]. In other words, the rapid public embrace and adoption of the smartphone has also had lasting impacts on society.

Another societal impact pertinent to this work is software dissemination models. That is, the mechanisms, policies and procedures governing the creation, publication, transportation and consumption of software. In this case, the dissemination (and profit) models proven viable by mobile devices are now increasingly adopted by PCs. Specifically, Microsoft's Windows, Apple's OSX, and various Linux distributions all embrace the notion of online marketplaces from which software is procured. This model sharply contrasts with traditional supply chains such as boxed software purchased from the shelves of brick-and-mortar stores.

Regarding software markets, the standard examples of two prevailing models are currently Apple's iOS ecosystem and Android's ecosystem. Briefly, the former is often categorized as a "closed" system where Apple maintains totalitarian control over all aspects of the ecosystem. Conversely the later is an "open" system allowing competition among markets. Indeed, many Android markets now exist. Amazon maintains its own Android Appstore as does Samsung. However, Google also maintains a market, Google Play, and clearly strives to maintain market leadership among Android users.

Between the Android and iOS models, to date, malware has clearly found more traction in the Android model. However, this disparity may not be inherent in the differences between the models. Contrarily, current tendency of malware to afflict Android may be caused by leading market share, vulnerable system architecture or implementation, lack of detection capability on iOS, and so on.

This dissertation supports the thesis that:

> *Using a combination of (1) application developer-oriented protections, (2) software market-oriented protections, and (3) verifiable software delivery, it is possible and easy to provide safety to users of software marketplaces, regardless of software dissemination mechanism.*

In other words, it is possible to perform safe software dissemination through digital application marketplaces that are distributed in nature. If true, end users can have confidence in the safety of their software and overall security posture of their devices. Ultimately, end users can be assured that software installed is that which the original publisher intended, regardless of how the software was obtained. Such assurance gives confidence to the consumer and enterprise alike.

1.1 Research Questions

In order to provide the desired software dissemination safety, we must first understand the problem space, the potential harm caused by malware. With this knowledge in hand, we can evolve the complementary solutions toward a safe dissemination model. To this end, we consider four research questions:

1. How prevalent is the problem of mobile malware?
2. Are there systematic issues brought on by mobile platforms that enable malware?
3. Are there new attack models that did not exist prior to the advent of the smartphone?
4. What can be done to mitigate the problem at pertinent junctures?
    a) How can mobile application developers create safer software?
    b) How can publishers safely disseminate software to end-users?
    c) How can market proprietors encourage safety in their marketplaces?
    d) How can end-users have assurance in the software security posture of their device?

1.2 Research Scope

Internet-based dissemination is rapidly becoming the dominate form of software delivery for all form factors of computing. Due to maturity and novelty, we look to mobile markets as exemplar instances of such dissemination. While such mobile markets have only existed for a few years, they are more mature than related markets, such as those found on the PC.

Fundamental to our desired goal of safe software dissemination regardless of distribution channel, Android's ecosystem is fundamentally distributed, allowing for secondary markets such as Amazon's Appstore. Ultimately, many users may never stray from original settings on a mobile device implicitly consenting or explicitly deciding to use the device's default software market. However, users may elect to use an alternative market or set of markets for many reasons. For instance, an Android user may install Amazon's Appstore application in order to gain access to

Amazon's streaming video service, which Amazon does not make available in Google Play. Conversely, a consumer that has purchased an Amazon Kindle Fire tablet, may stray from Amazon's Appstore in order to access the much larger library of applications available on Google Play.

Often compared to the Android ecosystem, especially in terms of security, is that of Apple's iOS. The iOS ecosystem is tightly controlled, from the hardware all the way to the application marketplace: the App Store. Users squarely place trust in Apple and Apple's ability to provide safety. In Apple's model, a single entity has total control of the system, in this way, the model is simple. However, such singular models also exhibit centralized points of failure. For instance, in November 2015, a signing certificate expired, leaving Apple OSX users with applications that they could no longer use [96]. The technical fix was straight-forward, but events like this highlight the unilateral simplicity of the model. Such simplicity also precludes much of the need for iOS model analysis.

In contrast to the iOS model, Android provides for much more interesting security and privacy decisions. Users may elect to pursue a model similar to iOS. By purchasing a Google Nexus device (where Google controls software updates) and electing to only install software originating from Google's Play market, the user can place immense trust in Google, similar to an iOS user's relationship with Apple. On the other hand, users may take a completely different stance, purchasing an Android-based device that uses no Google services and exclusively obtaining software from markets other than Google Play.

Between the two extremes of placing entire trust in Google and none, lays a continuum of variation. It is along this entire continuum that our research is focused.

1.3 Contributions

The Android application ecosystem is a complex model where developers push software to online software marketplaces, from which users user pull software. The actions in this model, publishing software and procuring software, are not initiated by markets, which instead act as central intermediaries for software dissemination. Miscreants may seek to violate security properties in development, publication, dissemination, procurement or use of the software. In order to provide safety in this ecosystem, we focus on three entities with clear equities: developers, market proprietors, and end users.

Developer-oriented protection Our motivation for developer-oriented protections is the observation that applications often request, at install time, many seemingly over-reaching application level permissions. The set of permissions requested is specified by the developer, and not only presents direct risk to the user (software that has access to inessential resources) but also forces the user to reason about an ill-informed, security-sensitive proposition: installing the application or aborting. We aid the developer by assisting in the specification of the minimum set of permissions, in turn reducing the set of permissions a user must reason when installing software.

Mobile Malware Measurements Based on a series of measurements, we establish patterns of growth in mobile malware, both in raw quantities, and in technological sophistication. We coarsely link the evolution of mobile malware to that of traditional malware (as found on the PC), and note that, compared to PC malware, mobile malware is not only evolving at a more rapid pace, but also already has complementary classes of malware as found on the PC. We provide measurements regarding the prevalence of malware across many software marketplaces, finding that some marketplaces are quite seedy, disseminating malware exclusively. Our measurements also reveal that the vast majority of mobile malware employs a common technique for grafting malicious software functions into existing mobile applications.

Evasion Analysis Contemporary malware has recently started to actively evade security analysis. Specifically, by detecting the presence of tools often used for analysis, the malware can alter its behavior to present benign operation when under analysis, and malicious operation otherwise. In order to best inform potential defenses, we generalized Android-specific evasions, ultimately developing and evaluating several unique methods of evading security analysis.

Market-oriented protection For markets that elect to police their software offerings (providing safety as a service to their consumers), we provide a novel mobile malware analysis platform. Our platform is designed with mobile-specific interactions and our more broad understanding of evasions in mind. Systems like ours may be employed by market proprietors to assist in policing their marketplaces.

Protocol-oriented protections Finally, we present an end-to-end verification protocol that binds developers to end users, providing assurance that software on the end device is that which the developer intended. Thereby providing safety regardless of how software is published, disseminated, and procured.

1.4 Thesis Structure

In Chapter 2 we provide background and security analysis of Android, architecturally and as an ecosystem. Here we detail fundamental security issues that enable malware and contrast Android's deviation from the familiar protections found on PCs.

The evolution of mobile malware and a combined analysis of mobile and PC malware are detailed in Chapter 3. In this chapter we also provide measurements, demonstrating the size of the mobile malware problem. Our measurements indicate that mobile malware is a growing concern. Additionally, observations of real malware dissemination inspire intuition toward potential methods of thwarting would-be attackers.

In Chapters 4, 5, and 6 we respectively turn to protections, methods of providing software safely. In turn, we address three stages of dissemination: providing safety as a developer, as a market proprietor and fundamentally in the form of a delivery protocol. Chapter 4 details a tool created for developers to minimize risk in the applications they create. Chapter 5 aims to provide market proprietors with the ability to police their market. Core to our thesis, Chapter 6 details a protocol designed to enable safe software delivery. We introduce the protocol, dubbed AppIntegrity, and show it to be consistent with current ecosystem practices. Further, we evaluate the efficacy of AppIntegrity via user studies that measure an exemplar interface. We provide conclusions and future research directions in Chapter 7.

Chapter 2

Android Security Model

In this chapter we focus on Android security. However, the topic of Android security is not a narrow field. Our aim is to provide a broad understanding of relevant aspects of the Android platform and the overall ecosystem that Android enables.[1] Building upon this understanding, we will turn to those taking advantage of security weakness in Chapter 3.

Today, smartphones have as much processing power and memory as a high-end laptop computer only a few years old. The additional capabilities of a smartphone also introduce a range of issues that are not present in either the security models for portable computers or traditional mobile phones. Modern smartphones are always-on devices which combine phone network connectivity with high-speed data networking capabilities and geolocation services (e.g., Global Positioning System (GPS)). Further, the vast array of mobile applications that extend the feature set of a smartphone enables the user to keep considerable amounts of (private) information on the device. All of these factors pose a host of security and management problems.

The current management models employed by the major smartphone systems make them, to a large extent, more similar to corporate-managed devices, rather than personal machines. For instance, phones are by default not "rooted" (i.e., the user does not have administrative privileges). This management model introduces security side-effects that users are unlikely to consider. Security issues that plague typical computing (e.g., privilege abuse resulting in unauthorized network access) now also apply to mobile devices. However, common security measures deployed on personal computers (e.g., rapid patch cycles and AV software) are made more difficult by the managed security model, leaving the user fundamentally unprotected.

Until recently, mobile devices used to present high Operating System (OS) heterogeneity, as each device had a rather unique OS variant. Such heterogeneity created a difficult management situation for cellular carriers, but also made the devices more difficult to attack, as a given vulnerability and subsequent exploit would apply to a relatively small set of devices. With Android, iOS and Windows Mobile now representing a substantial portion of the 64% of

[1]Portions of this chapter previously appeared in [230].

all Americans that own a smart phone (up from 31% in 2011) [10, 37, 207], the situation has markedly changed toward a more homogeneous deployment, which makes for more efficient management, while simultaneously facilitating large-scale attacks [117].

The ubiquity and increasing power of so many mobile devices poses a number of security challenges that must be addressed. In this chapter, we provide a taxonomy of mobile platform attack classes with specific, concrete examples as each class applies to the Android environment. Where possible, we also provide attack mitigations that can be added to the current security model. The rest of this chapter is organized as follows. In section 2.1, we provide relevant background of Android's security model and in section 2.2, we analyze new challenges that result from this model. In section 2.3, we detail specific attacks applicable to different parts of Android. These attacks ultimately provide privileged access for a determined adversary. In section 2.4, we provide possible mitigations for some of the attacks considered. We offer concluding remarks in section 2.5.

2.1 Android Security Model

Android was created with certain security design principles, such as privilege separation, in mind [203]. At its core, Android is a Linux-based open-source OS, with a layered structure of services [32] including core native libraries and application frameworks. Android natively separates applications and provides safety through OS primitives and environmental features such as type safety [203]. At the application level, each software package is sandboxed by the kernel, making Android a widely deployed system that employs privilege separation as a matter of course. This sandboxing is intended to prevent many types of information disclosure such as one application accessing sensitive information stored on the system or in the private space of another application, performing unauthorized network communication, or accessing other hardware features such as the camera or GPS. Applications instead request access to system resources via special application-level permissions, such as `READ_SMS`,[2] which must be granted by the user.

Android's permission model requires each application to explicitly request the right to access protected resources. The permission model is intended to prevent the unwarranted intrusion by an application on the user's data and the data of other applications, as well as limit access to features that directly or indirectly cause financial harm (e.g., a mobile phone plan that charges for each Short Message Service (SMS) message). In versions of Android prior to 6 (Application Programming Interface (API) 23, i.e. "Marshmallow"), before installing an application, the user is presented with a list of all[3] the permissions requested by that application which they must accept before installation

[2]For all permissions, we truncate the longer notation throughout this text for brevity (e.g. `android.permission.READ_SMS` to simply `READ_SMS`).

[3]We show later that this list is not necessarily comprehensive.

begins. Not only is the permission set so confusing that it is difficult for users to read and understand, but it is also a binary decision where the user must accept all requested permissions or not install the application [203].

The semantics of these application level permissions may be difficult for most users to understand [61, 227], but this understanding is further complicated with implementation details and design choices. For example, a method of inter-process communication employed by Android known as *Intents* can be broadcast to several software handlers existing in multiple applications. Each application has the option of registering a handler for an Intent, and for some Intents, an associated priority. When pondering permissions at install time, a user has no way of knowing what priorities are assigned to a given handler or what effects installation will have in conjunction with already installed applications when a particular event occurs.

Applications are made available to users through an unmoderated venue, Google Play (formerly the Android Market). Any entity can create an Google Play account for a modest fee, and immediately make software available to the public. While mobile applications on Android must be signed, they are typically self-signed without employing any kind of central authority or any means for a user to validate the authenticity of a certificate. This open policy can actually make the market a means for propagating malware [168]. Attackers can easily publish malicious applications to a venue with over 50 billion downloads [236], needing only an appealing facade to convince users to install the malicious code. For example, BadNews malware in 2013 infected millions of Android users [187]. In this case, the facade consisted of at least 32 applications published by four Google Play developer accounts. All of these applications employed a fabricated advertising network meant to camouflage the delivery of malicious content to infected devices.

The open nature of the Google Play requires a management model that facilitates reactive malware management after applications have been installed. The malware issue is currently addressed with remote [un]install capabilities maintained by Google via the Google Services software present on every Android device. Following the identification of malicious applications, possibly through Android users rating an application negatively, Google may remove applications from the market and remotely kill and uninstall the applications from devices known to have downloaded the application [168]. Interestingly, while Android is largely considered to be open-source, the sources for software such as Google Talk, Google Services, the Google Play app (Vendor.apk), and carrier-specific update software are not readily available. In this model the user only maintains control of applications found in the restricted user space, and even this can be over-ridden via remote uninstall.

2.2 Android Security Model Analysis

As Linux provides security-oriented facilities such as process separation, file system access controls, and User Identifiers (UIDs), much of Android's on-device, architectural security is bound to Linux. As such, many Linux vulnerabilities also afflict Android. Even with the inclusion of security as part of the original design, the new security features create new opportunities for attack, and the growth of the platform provides incentive. In this section we discuss several features of the Android model that may make it vulnerable to attack.

2.2.1 Application permission model

Unprivileged application attacks can take advantage of the complexity of the Android permission model and the Google Play to persuade users to install malicious software and grant applications the permissions required to deliver a harmful payload. Applications advertised in Google Play must be signed by the author, but the user has no means of verifying that a signature is associated with any particular entity [99].

Even users that carefully monitor permission requests may not fully understand the semantics of a particular permission or the framework behavior surrounding a permission. Many typical users may simply not understand the security trade-offs that surround permissions like `ACCESS_SURFACE_FLINGER` or `BIND_APPWIDGET`. Other permissions, such as `SEND_SMS` are more likely to be recognized by users, and yet the framework implementations may cause undesired behavior. For example, the receipt of an SMS on a device causes an Android *ordered-broadcast* to be sent system wide. Applications can register the ability to take action when the broadcast is observed by the application and can request any valid priority over the broadcast. Thus, an application that registers a higher priority will receive, and have the ability to act upon, this broadcast before any other application at a lower priority.[4] In regard to SMS, this means that any application with a priority higher than the standard SMS application will have the ability to process the incoming SMS message first, including the ability to prevent the broadcast from being observed by any other application. Malware employing covert communication channels in this manner has already been seen [126,223]. When a user grants permissions based on prompts that show a `RECEIVE_SMS` permission, such behavior is likely unexpected.

Similarly, the only means a user has of understanding the capabilities of an application is through the list of permissions presented at install time. In addition to the framework implementation issue mentioned above, additional ambiguity is introduced via new permissions that exist in later versions of the Android API. For example the `STORAGE` permission relates to accessing an external Secure Digital (SD) Card. The `STORAGE` permission was not available until Android 1.6 (API 4), so the behavior of an older application on a newer device is ambiguous. Current devices simply do not display a comprehensive list of permissions to the user. An older application that does not request the `STORAGE` permission, and should not as the permission is not defined for the older API, will still implicitly have access to the SD Card [22].

In effort to address poor user understanding, permissions were later grouped, and these groups were named in seemingly intuitive ways. For instance, instead of inferring the meaning of the application-level permission

[4]Applications at the same priority receive the broadcast in an order undefined by the Android API [3].

Protection Level	Value	Description
normal	0	A low-risk permission representing little risk to other applications, the system or user. Applications that request access to these permission are automatically granted access.
dangerous	1	A high-risk permission that may grant access to user data or exercise control over the device. Android may not automatically grant access to applications that request access to these permissions.
signature	2	A permission that the system will only grant if the requesting application is signed by the same certificate as the application that declared the permission (for correct certification matches, the permission is automatically granted).
signatureOrSystem	3	The same as "signature" while additionally automatically granting access to Android system image applications.

Table 2.1: **Android permission protection levels**, there are four fundamental protection levels as of Android 6 (API 23) [3].

`ACCESS_FINE_LOCATION`, the user may be prompted with "Allow Twitter to access this device's location?" Google partitioned application-level permissions into protection levels (Table 2.1), many of which have existed since Android 1 (API 1), but the semantics around these levels changes substantially in Android 6 (API 23). Following API 23, for permissions considered *dangerous*, the permission is granted explicitly by the user upon first access by the application. For example, the first time an application attempts to use `ACCESS_FINE_LOCATION` the user is prompted with a modal dialog to approve or reject access to the associated permission group `LOCATION`. This preference may be stored so the user is not prompted for subsequent access to the same resource from the same application. At any time, the user can access advanced settings for any application and manipulate access to any of the supported permissions on a per-application basis.

On the surface, this design seems appealing, more users may take the time to read the short prompt and consider the meaning of "location." However, the groupings actually cause the user to make very coarse decisions, removing more granularity from the already lacking permission system. In the case of `LOCATION`, the user is unable to distinguish between coarse and fine location. Users that prefer to have some applications access network-based location at less precision than GPS have no way to differentiate. Similarly, `SMS` simultaneously grants read and write permissions—there is no mechanism to permit receiving text messages without also granting the ability to send. `PHONE` groups the ability to read recent call log entries with telephony operations. Some groupings may even be nonsensical to users, such as the combination of all Camera operations (picture, video) with the device's microphone—prior to API 23, a voice memo application would be granted access to record video, in addition to the microphone. Also, with the introduction of these permission sets, the `INTERNET` permission is now silently permitted for all applications.

Of the protection levels, of particular interest are *normal* and *dangerous* permissions, outlined in Table 2.2. Google subjectively performed this partitioning under the guidance that granting a normal permission is of "no great risk to the user's privacy or security [19]." As an example, "users would reasonably want to know whether an application can read their contact information, so users have to grant this permission explicitly. By contrast, there's no great risk in allowing an application to vibrate the device, so that permission is designated as normal." On the other hand, permissions with

Protection Level	Permission Group	Permission
dangerous	CALENDAR	READ_CALENDAR WRITE_CALENDAR
	CAMERA	CAMERA
	CONTACTS	READ_CONTACTS
		WRITE_CONTACTS
		GET_ACCOUNTS
	LOCATION	ACCESS_FINE_LOCATION
		ACCESS_COARSE_LOCATION
		MICROPHONE
		RECORD_AUDIO
	PHONE	READ_PHONE_STATE
		CALL_PHONE
		READ_CALL_LOG
		WRITE_CALL_LOG
		ADD_VOICEMAIL
		USE_SIP
		PROCESS_OUTGOING_CALLS
	SENSORS	BODY_SENSORS
	SMS	SEND_SMS
		RECEIVE_SMS
		READ_SMS
		RECEIVE_WAP_PUSH
		RECEIVE_MMS
	STORAGE	READ_EXTERNAL_STORAGE
		WRITE_EXTERNAL_STORAGE
normal	n/a	ACCESS_LOCATION_EXTRA_COMMANDS
		ACCESS_NETWORK_STATE
		ACCESS_NOTIFICATION_POLICY
		ACCESS_WIFI_STATE
		BLUETOOTH
		BLUETOOTH_ADMIN
		BROADCAST_STICKY
		CHANGE_NETWORK_STATE
		CHANGE_WIFI_MULTICAST_STATE
		CHANGE_WIFI_STATE
		DISABLE_KEYGUARD
		EXPAND_STATUS_BAR
		FLASHLIGHT
		GET_PACKAGE_SIZE
		INTERNET
		KILL_BACKGROUND_PROCESSES
		MODIFY_AUDIO_SETTINGS
		NFC
		READ_SYNC_SETTINGS
		READ_SYNC_STATS
		RECEIVE_BOOT_COMPLETED
		REORDER_TASKS
		REQUEST_INSTALL_PACKAGES
		SET_TIME_ZONE
		SET_WALLPAPER
		SET_WALLPAPER_HINTS
		TRANSMIT_IR
		USE_FINGERPRINT
		VIBRATE
		WAKE_LOCK
		WRITE_SYNC_SETTINGS
		SET_ALARM
		INSTALL_SHORTCUT
		UNINSTALL_SHORTCUT

Table 2.2: **Android application-level permissions**, protection levels normal and dangerous, along with permission groups associated with dangerous level, Android 6 (API 23) [3].

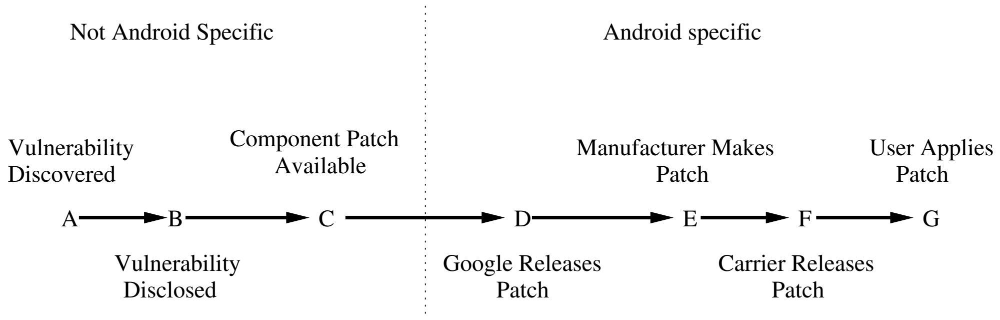


Figure 2.1: **Android patch cycle**: Lifecycle of an Android patch from vulnerability identification until a patch reaches the user device. Figure from [230].

more perceived risk are designated as dangerous. Normal permissions are granted to applications implicitly, as there is no perceived risk. Of the 136 standard permissions[5] defined for API 23, 36 are labeled normal, and 23 of are labeled dangerous, and the remaining permissions are largely not intended for use by third-party applications (i.e., internal use only).

2.2.2 Patch cycles

In an effort to increase adoption of the Android OS, Google created the Open Handset Alliance to build cooperation between hardware manufacturers to facilitate implementation on their devices [21]. Through this cooperation, Google provides the base open source operating system, then device manufacturers and telecommunications carriers modify this base to differentiate their offerings from other Android devices and provide added value to their customers. Figure 2.1 shows the patch cycle from a vulnerability's initial discovery until the patch eventually reaches the user's Android device. Patch cycle events A through C are typical of any software product. The software fix represented by C is typically the end of the vulnerability window introduced at A. When considering one of the re-used components in Android, such as WebKit, D-G are appended to the patch cycle. The additional states added to the cycle come from Google's cooperation with multiple manufacturers and carriers. Whenever a patch to Android becomes necessary, Google provides an update through their open source forum and manufacturers then proceed to port the update to their customized version of the operating system.

Google's historical major release cycle is approximately four months, with minor patches released intermittently [3] (depicted visually later in Figure 3.1). However, this is not the date on which these updates are actually available to users. Once Google releases its patch, the manufacturer must then update it to work with their custom hardware [71]. These updates may actually never be made available to the user if the carrier deems deployment too costly [242]. After the manufacturer modifies the patch to work with their custom device, there can be a delay before

[5]This does not include other system permissions, such as those with Protection Level "signature," or non-system permissions that may ship with Android, such as `com.android.browser.permission.READ_HISTORY_BOOKMARKS`.

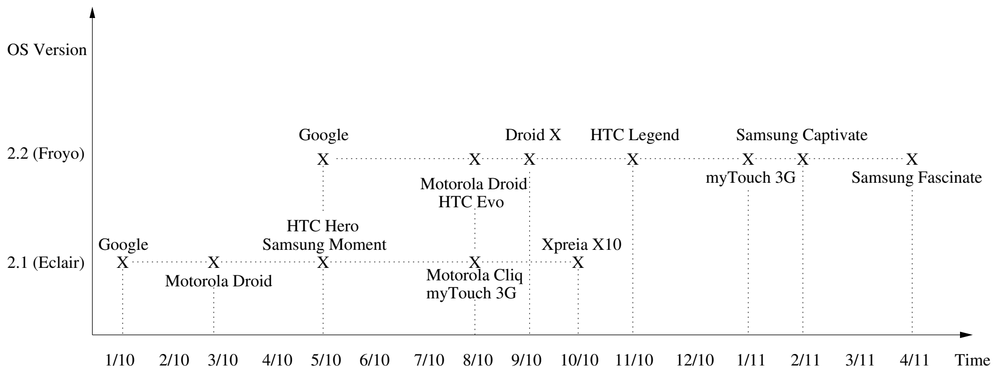


Figure 2.2: **Android device update timeline, by version**: Google [3] and manufacturer releases of Android 2.1 [87, 122, 125, 241] and 2.2 [184]. Figure from [230].

the patch is released by the carriers. An example of a delay between the manufacturers release of a patch and the patch's eventual release to the public can be seen in [156] where AT&T postponed the release of Android 2.2 to the Dell Streak creating a three month gap between states *E* and *F* in Figure 2.1.

Examples of the disparity between Google's release of Android updates and their eventual release to specific devices can be seen in Figure 2.2. It is not uncommon to observe at least two months (and sometimes much more) of delay between an Android update and an actual deployment of the update by the major manufacturers. As a case in point, Android 2.1 only became available on certain devices (e.g., Xperia X10) after Google had already superseded it with Android 2.2.

Android 2.1 was released in January 2010, however it was another two months (March 2010) before Verizon pushed this update to customers using the Motorola Droid. It was yet another two months (May 2010) before other popular devices, HTC Hero and Samsung Moment, received the update. Some devices, such as the T-Mobile myTouch 3G and the Xperia X10, even received the update after Google published version 2.2. This slow patch time is not specific to the 2.1 update. As of April 2011, approximately 33% of all Android devices currently in use had yet to receive the update for Android OS 2.2 "Froyo" [24], almost a year after the updates original release. For example, the Samsung Captivate did not receive the 2.2 update until February 2011, 9 months after the major version release.

Similarly, these update delays are not an artifact of the past. Recent studies indicate that the patch delays result in a substantial percentage (about 25-30) of devices that may not receive security updates until after a new vulnerability is discovered [216]. Generally, the security posture of these insecure devices is no different than if they never received security updates. Figure 2.3 shows the proportion of insecure devices (red) over time. The proportion of non-red rarely enjoys majority.

With such a large vulnerability window and the separate release dates for Google and each manufacturer, attackers can use reverse engineering techniques to identify and exploit a vulnerability on a device using information available in the original released patch or any other manufacturer that adopts the patch earlier [68]. Platforms that regularly

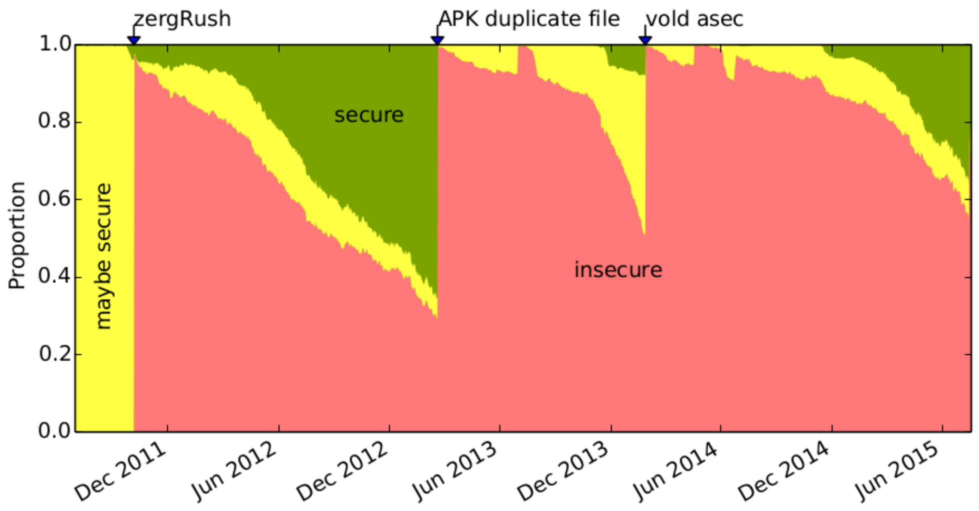


Figure 2.3: **Proportion of Android devices with insecure software**: According to 11 vulnerabilities detailed in [216], the proportion of measured devices susceptible to at least one vulnerability over time. The gradual decline in red (insecure) reflects the application of patches. The sharp increases in red are when new vulnerabilities are discovered. Figure from [216].

exhibit slow patch cycles are at even greater risk since the patches made available to comparable systems can be analyzed in order to determine the vulnerable conditions making the still un-patched devices easy targets.

Because of Android's slow patch cycles, the standard re-use of common "upstream" software components that underline the Android framework can cause increased vulnerability. Within the Android framework, common open-source components such as WebKit and the Linux kernel are re-used to reduce the cost of system design. Component re-use is a common practice among large systems such as Android. Apple's iPhone similarly employs WebKit [5] and a Berkeley Software Distribution (BSD) kernel derivative (Darwin) [30]. The re-use of common software components itself is not detrimental, but the additional delay introduced by Android's patching model is. Once a vulnerability in WebKit or Linux is discovered, it is generally patched and released quickly by the open-source community, however the corresponding Android patch may not be available to users for months. The imbalance between the Android patch cycle and the software components it is built upon leaves a longer attack window for Android devices. Similarly, due to Android's Linux foundation, lower level attacks are simpler when compared to less ubiquitous OSes, as attackers do not need to learn a new kernel [47]. For Android to provide a secure platform it must not only strive for a secure framework, but also provide timely updates to minimize the attack window. Figure 2.4 depicts several upstream projects and the quantity of patches effected between each Android-specific party (D-G in Figure 2.1).

2.2.3 Trusted USB connections

With local USB access, a developer can interact with an Android Debug Bridge (ADB)-enabled device. The ADB is a development tool provided by Android intended to gather debugging information from an emulator instance or

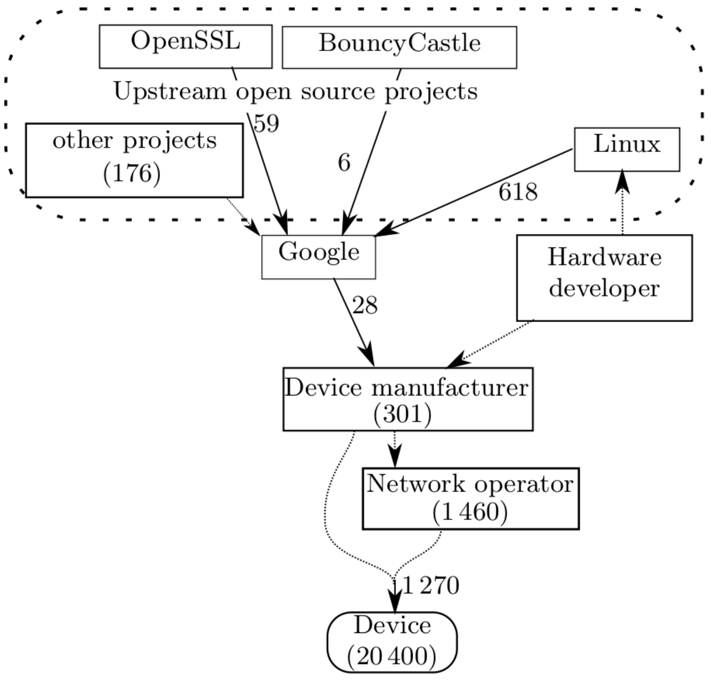


Figure 2.4: **Android 'upstream' software patch cycle**: Lifecycle of an Android patch from patch issue on upstream projects until a patch reaches the user device. The edge weights are updates shipped between July 2011 and July 2015. The dotted arrows indicate flows that were not measured. Figure from [216].

USB-connected phone [3]. The ADB provides further benefit to the user by facilitating direct installation and removal of applications, bypassing Google Play, and providing access to an unprivileged interactive remote shell. Using the ADB interface, attackers can add and execute malicious tools that exploit vulnerabilities in the device. Such malware does not need to be present in Google Play and will not be tracked for possible remote uninstall by Google. With ADB access an attacker can again take advantage of Android's slow patch cycle to gain privileged access to the device. The ADB can be used to simply execute command or extract information and the ADB can be used while a device's screen lock is active.

At the onset of our research, the ADB was unauthenticated and simple to enable on devices. Starting with Android 4.2 (API 17), Google made it more difficult to enable the ADB on devices. Instead of having a toggle in relatively intuitive location in Device Settings, users must now perform unobvious actions to enable an unadvertised developer menu. Specifically, the user must navigate to "Settings -> About phone" and tap the "Build Number" seven times. At which point, a new menu item, Developer Options, appears on the Settings screen. Following this hindrance, a form of authenticated ADB was introduced in Android 4.2.2 (API 17). After 4.2.2, instead of simply allowing interactive access to connected devices, a prompt is displayed asking whether the user accepts a particular Rivest-Shamir-Adleman (RSA) key associated with the connected computer. Authenticated ADB is explored further in section 2.4.5.

2.2.4 Recovery mode and boot process

Android devices employ a special recovery boot mode that enables device maintenance. The recovery mode allows the user to boot to a separate partition on the device circumventing the standard boot partition [231]. Android's default recovery mode image facilitates maintenance features such as applying a system update to standard system and data areas, cleansing a device of user data ostensibly for re-sale, or software repair to restore life to corrupted (or "bricked") devices. Because there is no trusted component to the boot system, attackers can utilize the separate recovery partition by loading in their own malicious image to gain privileged access to the user's information without affecting user data.

In recent years some manufacturers have "locked" bootloaders on various devices (and some have subsequently reversed their stance on locked bootloaders). For the most part, a locked bootloader means software running on the device will not permit replacement bootloader software updates unless they are properly signed by the manufacturer. The assertion is that this provides security to the user (i.e., nobody has undermined the security of the device while it was left unsupervised at a coffee shop) and also protects the manufacturer or cellular provider (who maintain more control over the device). Locked bootloaders can present a serious obstacle to those that wish to install alternative software on the device.

Starting with Android 4.4 (API 19), optional support was added for "boot verification." Under this paradigm, devices may be in two states: locked or unlocked. Locked devices will only boot if the boot partition cryptographically verifies with an Original Equipment Manufacturer (OEM) signing key. If the device is in unlocked state, or the boot partition fails verification, various warnings or notifications are presented to the user depending on circumstances. Each of these warnings requires the user to acknowledge the warning, or, if there is no user interaction the warning will be silently bypassed after a timeout of at least five seconds.

2.2.5 Uniform privilege separation

Users can download applications marketed for security, however the effectiveness of such applications is extremely limited. On a typical computer, security software, such as AV, may require privileged access in order to secure permission to scan all files. Security applications on Android are limited to the same restricted environment as every other Android application. Even applications that claim to "block malware, spyware and phishing apps" require a rooted device in order to obtain the necessary permissions to perform these functions [16].

Likewise, most Android applications that claim firewall capabilities only provide call and SMS filtering. Indeed, adequate network firewall features require a rooted device [203]. Without security tools available, users cannot identify malicious content and the burden falls on the device manager.

2.3 Attack Classes

Perhaps unsurprisingly, full privileged device access is generally guaranteed given physical access. Perhaps less surprising is the relative ease with which privileged access can be obtained and the amount of nefarious activity that is possible *without* privileged access. In this section we describe unprivileged malicious applications and the circumstances and methods necessary to gain privileged access to an Android device with respect to the following attacker capabilities.

No physical access: Attack circumstances where it is impossible to gain physical access to a user's device. Then the attacker must get the user to perform actions on the attackers behalf. Such remote attacks commonly rely heavily on social engineering [35]. To achieve the appropriate initial access to the user's device an attacker must achieve malicious software running on the device. To execute code remotely on a user's device, the attacker typically must convince the user to either download a malicious application or access malicious content via one of the applications already installed on the device. If the attacker can exploit a vulnerability on the user's device, then this access may be used further to gain privileged access.

Physical access with ADB enabled: If the attacker finds a device left unattended, yet obstructed via a password, Personal Identification Number (PIN), or pattern-lock, the attacker may be able to exploit the device through the ADB. This method is more difficult on devices with Android 4.2.2 and later.

Physical access without ADB enabled: If the attacker finds an obstructed Android device left unattended, but is unable to use the ADB service, the attacker may still gain privileged access via recovery boot. This method is more difficult on devices that employ locked bootloaders.

Physical access on unobstructed device. In some cases the attacker may actually have access to a device without a password protected screen lock. Such a situation allows the attacker to actually leverage any other attack method since the attacker can choose to install applications, visit malicious websites, enable ADB on the device, etc.

We next detail specific attacks that demonstrate the real threats present in the cases enumerated above. Note that we do not include physically destructive attacks such as opening the casing in order to access debugging ports (e.g. Joint Test Action Group (JTAG)). The last case is included for completeness, but not specifically detailed since all of the mentioned attacks would also be applicable in the case of physical access to an unobstructed device.

2.3.1 Unprivileged Attacks

Much like a user that will install an application insecurely downloaded from the Internet despite any OS warnings, a user may easily install applications that request dozens of Android permissions without a second thought. Applications that are restricted via Android's typical application sandbox, but that have copious access to resources through permissions, can perform many of the same functions of malware common on the personal computer platform [97].

For example the Zeus botnet architecture has been ported to most mobile platforms [34], worms such as Yxe have been seen on SymbianOS [36], and trojan malware has been found in applications present in the legitimate Google Play [29, 99]. Users that completely disregard, or are tricked into accepting, prompts from the device regarding install time permissions effectively permit negative actions completely at the application level.

In some cases, misleading the user may not even be required. Some software handlers (Android *Receivers*) may be registered by an application at install time. Intents expected to occur regularly or those that can be remotely invoked can be used to achieve remote code execution without requiring user action. Users may install applications from the Google Play web interface, which subsequently initiates a remote install. Thus, an attacker that has somehow obtained a user's authentication token to the web interface can remotely install applications to a user's device. To achieve remote code execution the attacker need only perform an action that results in the device generating an Intent that the app for which the application has a receiver. Even security-related features, such as the screen lock may be remotely bypassed, by registering a receiver (for example `PACKAGE_ADDED`) and using the legitimate API for `KeyguardManager` to disable the screen lock upon application installation [72].

Application *re-packaging* (explored in more detail in section 6.1) has proven to be an effective means to entice users enough to download malicious applications. Entire families of trojan software have been classified in third party black markets as well as Google Play. Families such as Geinimi [181] and DroidDream [150] have been found in dozens of applications in the Google Play. Many of these applications offer no additional value to the consumer, they are simply existing, popular applications that have been reverse engineered to the point that the attacker can augment the existing application with the malware and re-package the application. The attacker then signs the new application with a new key and makes the application available in a market for victims to download.

A new application attack does have considerable drawbacks in that the user can often observe the existence of the application at any time via the application manager, and may even chose to uninstall the application. Certain application actions may also cause undesirable indicators to show, such as a GPS notification icon, or may even interact unfavorably with other installed applications.

The fact that malware can successfully operate within Android's restricted application environment greatly lightens the attacker's burden to achieve privileged access through escalation. Applications executing in an isolated environment simply obtain the appropriate permissions from the user at install time and perform nefarious activity from within the confines of the application sandbox. Even so, attackers may wish to obtain elevated privileges in order to perform actions for which no application level permission exists, to prevent undesired indicators, or in order to maintain guaranteed persistence.

2.3.2 Remote Exploitation

When turning to privileged access, an attacker may rely on convincing the user to install a malicious application. Such an application may present an enticing feature to the consumer but contain software that executes a privilege escalation attack. Oberheide [168] demonstrated such an attack. Oberheide's seemingly benign application received more than 200 downloads within 24 hours [168]. In the background, this application would routinely make remote requests for new payloads to execute. Whenever a new privilege escalation exploit was discovered for the current running version of Linux, the application could retrieve the exploit from a remote server in order to gain root access. These exploits must be delivered to the application before the vulnerability is patched, which is generally easy to do considering the long patch cycles previously discussed.

Similar privilege escalation methodologies can be found in legitimate device "rooting" applications such as Root Tools [25], Easy Root [9], and Unrevoked [243], which take advantage of vulnerabilities in the phone to gain privileged access. Users can then use this access to circumvent carrier controls over the use of a specific software or upgrade to a newer version of Android that their carrier has yet to release [231].

Deploying malicious applications with a benign facade through Google Play takes advantage of Google's reactive philosophy toward malicious applications. Once on Google Play, the attacker's application can now reach a global audience. Because there is no screening of applications, attackers are given a pedestal from which to entice the user.

A remote attack may not even require the installation of a new application. Android's use of commodity software components, such as the web browser and Linux base can be leveraged for an attack that requires no physical access [203]. A concrete example of such an attack was deployed by Immunity Inc. for the penetration testing tool CANVAS 6.65 [47]. In this attack, the user visits a malicious website using the device's built-in browser. The attacker then uses this request to take advantage of a vulnerability in the WebKit browser (Cascading Style Sheet (CSS) rule deletion vulnerability [244]) to obtain a remote shell access to the device with the level of privilege granted to the browser. The attacker can then copy a Linux privilege escalation exploit to an executable mount point on the device, run the secondary exploit, and gain privileged access to the whole device.

Another remote surface is the cellular baseband, the technical interface that facilitates connection to cellular towers. Baseband software is typically updated independently, but in conjunction with general device firmware. Remote attacks, with with local proximity, may be performed to take advantage of vulnerabilities in baseband software or protocols. Attackers might leverage a flaw in the software to achieve code execution or might impersonate cellular towers or software update providers to a target device.

2.3.3 Physical Access with ADB Enabled

Similar to the previous case, it is possible for an attacker to obtain privileged access through physical access to a device that has ADB enabled [40]. Given physical access, an attacker can easily determine if ADB is enabled, by connecting the device via USB and executing `adb get-serialno` on the attached computer. If the device's serial number is returned, then ADB is enabled. On devices with authenticated ADB enabled, a serial number consisting entirely of question marks (????????????) is returned.

Once the attacker knows that ADB is enabled on the device, a privilege escalation may only require the attacker to use ADB's `push` feature to place an exploit on the device, and use ADB's `shell` feature to execute the exploit and gain privileged access.

Different from many remote attacks, an attack on an ADB-enabled device does not require any action from the user. Privilege escalation using ADB does rely heavily on the availability of an enabled debug bridge, which is often only found on devices used by developers and not the typical user. However, if the device is not obstructed, the attacker could simply interact with the common device interface and enable ADB.

An example usage of this method for gaining privileged access is the Super One-Click desktop application [234]. Super One-Click requires users to first enable ADB debugging in the device. Once enabled, the application exploits the device using the previously defined method to grant the user privileged access.

The main advantage of ADB-based attacks is the minimal observable footprint left on the device: no new application must be installed on the device and a reboot is not necessary. The lack of device modification in this method makes it much more difficult for the average user to detect than other attacks, and is unlikely to be detected by security applications on unrooted devices.

2.3.4 Physical Access without ADB Enabled

On an obstructed device that does not have ADB enabled, the attacker may still be able to leverage flaws in the software or take advantage of fundamental architectural features, such as the device's recovery mode [231].

Absent any known vulnerabilities in the OS, an attacker may elect to pursue gaining access via a special boot mode. Since the attack does not rely on a software vulnerability specific particular to a version of Android, the attack has more longevity than other exploits such as the WebKit and Linux exploits mentioned above. The deployment method is device-specific leading to extensive fragmentation based on device model and/or manufacturer.

To use the recovery mode, the attacker must first create a customized recovery image. The main modification necessary for this image is to the `init.rc` and `default.prop` files in the `initrd` section of the image. To give the attacker the necessary privileges, the `init.rc` file must list the executables that they wish to add with the rights necessary to be executed. Any executables necessary for the attack must also be added to the `initrd` section of the image.

Once the image is built and the attacker is able to gain physical access to the device, the attacker must then attach the device to a computer through a USB connection and run a manufacturer specific tool (e.g., fastboot for HTC, RSDLite for Motorola, Odin for Samsung) to flash the image to the recovery partition of the phone. After the device has been flashed, the attacker then can access the recovery image using a device specific key combination (e.g., Power button while holding "X" for Motorola Droid). When the device loads into the recovery partition the `init.rc` file is executed. `init.rc` can be modified to run any malicious code added to the recovery image by the attacker, such as auto-installing a rootkit without attacker interaction. Alternatively, the attacker could update the `default.prop` file to enable ADB, crafting `init.rc` to give executable rights to an `su` executable (added previously to the custom recovery image `initrd`). When the recovery image loads the attacker opens an interactive shell on the device using ADB. The attacker can now simply execute the `su` executable to gain root access.

Installing a malicious recovery image takes advantage of the absence of any trusted boot system on Android systems, or in the case of devices that support boot verification, the absence of required acknowledgment or authentication. In this way, it is possible to make changes to the devices boot image and gain privileged access without the need to provide any authentication to the device.

If an attacker is able to create a custom malicious recovery image, it is feasible for this attacker to gain privileged access on any device for which physical access can be obtained. Installing a new recovery image and rebooting the device is not a perfect vector as it can leave an extensive footprint sans subsequent attacker cleanup. However, the recovery image replacement has negligible effect on the user's experience. While an attacker would almost certainly have a malicious payload, this technique is similar to methods described in [231].

2.3.5 Physical access to unobstructed device

If the device user has elected to not employ any kind of obstruction whether or not ADB is enabled on the device is irrelevant, as any of the techniques described above are possible if physical access to the device is available. The attacker need only turn the device on, download their malicious code, enable ADB on the device, etc. The method of attack that the attacker chooses now depends upon other metrics, such as covertness. For example, a malicious application remains observable by the user and may potentially be uninstalled in the future.

Obstructed Android devices still allow limited access to some resources, which complicates software security controls. In June 2015, a fundamental flaw in Android's lockscreen implementation allowed local attackers to circumvent password protection (afflicting Android 5 (API 21) < 5.1.1). In this attack, the miscreant would access the "emergency call" dialog present on mobile phones for 911 phone access. The graphical "cut-and-paste" capability can be used to copy a few characters, repeatedly pasting them until the text for the number to be dialed is so long, it can no longer be selected. At this point the copied text is very large in the device "clipboard." Any local user may access the camera application (even when the device is obstructed), however, accessing the camera settings requires authentication, When the authentication prompt appears, the clipboard text can be pasted until the Android user interface crashes, rendering access to privileged interfaces [131].

2.4 Mitigations

We originally proposed six possible mitigations [230] to the attacks seen in the previous section. Perhaps in light of our proposals, some of these mitigations have since been employed by Google and the Android Open Source Project (AOSP). Our proposals include reducing the patch cycle for Android updates, creating a trusted class of applications with privileged access, enabling authentication on Google Play downloads and the ADB, leveraging existing host security technologies, and deploying a Trusted Platform Module (TPM) on Android enabled devices.

2.4.1 Reduce the Patch Cycle Length

In all but one of the attacks shown in section 2.3, attackers exploit some flaw in the operating system to gain root privileges. Reducing the patch cycle length would mitigate these threats with greater effectiveness. Zero-day exploits would still be possible, however the common, lingering threat will be reduced.

While Google has already demonstrated willingness to act quickly with out of band patch releases in reaction to certain attacks [168]), reducing complete patch cycles is a more difficult problem. Indeed, manufacturers make changes to the Android source to create a competitive advantage. To reduce patch cycles, manufacturer modifications should not fundamentally change the core components of Android, and thus should not require a lengthy time to port the patch. A fundamental separation between the core of Android and manufacturer modifications should be established.

Even though the patch complications visible in Figure 2.2 are present on most Android devices today, some efforts have started to address the issue. Nexus devices have long enjoyed updates directly from Google, and in Fall 2015, the updates have moved to a monthly cycle—an unprecedented speed of updates for mobile devices [149]. Shortly thereafter manufacturers LG and Samsung also committed to monthly updates [84]. However, HTC claimed that monthly updates are unrealistic, chiefly because cellular provider testing programs cannot maintain such a pace [217]. So, while some have committed to substantially faster security update cycle (at least four times as fast previously achieved), it seems that, at least for now, the only beneficiaries are likely owners of Google Nexus devices and perhaps carrier-unlocked devices for some manufacturers. It is important to note, that striving for a monthly update cycle is somewhat "behind the curve." While the update paradigm is not quite comparable to that of Android, Microsoft's "patch Tuesday" for Microsoft Windows has become a common term in security communities as the monthly patch cycle has existed for more than a decade. Even so, in 2015, Microsoft indicated that it will start moving to a more frequent, intermittent patch cycle [163].

2.4.2 Privileged Applications

Many solutions have been proposed to mitigate application attacks that take advantage of Android's permission model. Barrera et al. suggest a restructuring of Android permissions into a hierarchy to allow for finer grained permissions that can be simpler for users to understand [61]. Under a hierarchy, an application displaying advertisements that would traditionally require `INTERNET` permission would need to have `INTERNET.ADVERTISING.adsite.com` permission, which would limit its connection to a specific site and let the user know exactly for what the permission is used.

In [99] Enck et al. propose lightweight application certification comparing the requested permissions of an application to a set of security rules. If the application does not pass any of the security rules, then possible malicious activity is brought to the attention of the user. Adding a certification mechanism to the Android framework would require modification of the Android security framework.

Android's application model could also be adjusted to allow certain applications to obtain additional, privileged device access. For example, Google could validate that certain software vendors create security software and grant applications created by these vendors additional API functionality. Applications signed by such a vendor could, for example, have read access to the filesystem in order to facilitate AV scanning beyond limited scope typically granted to applications. Such a configuration would allow users to install security related applications without having to first root their device. Because privileged applications will have less restricted access to the device, these applications should be certified by some governing entity before they can be downloaded. This certification process could also help mitigate some weaknesses of an unmoderated market. With access to trusted security tools, users would be able to monitor untrusted applications and provide appropriate feedback. With a market model split into trusted and untrusted applications, Google could provide enhanced security with minimal administrative overhead and minimal reduction in the openness of the platform.

2.4.3 Leveraging Existing Security Technologies

There are several existing operating system security enhancements that could be ported to Android. In [202], Shabtai et al. experiment with adding SELinux to Android. An information-flow tracking system, TaintDroid, has been created by Enck et al. instrumenting Android [97] to monitor applications and understand how they interact with the user's sensitive information. A realized implementation of TaintDroid could gives users real-time information about how an application uses the permissions it is granted. Generally, operating system level software modifications such as adding a firewall or SELinux to Android involve porting existing technology to the Android kernel and creating an application to facilitation administration.

In recent versions, Android has implemented existing technologies such as SELinux and Address Space Layout Randomization (ASLR). However, the implementations are still relatively immature and not applied uniformly to all devices. Further, many typical tenants of network and host security, such as network firewalls and IDS systems are still incompatible with today's mobile devices.

2.4.4 Authenticated Downloads

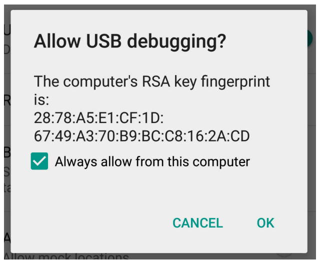
Once an attacker has physical access to a device, adding malicious applications becomes simple and quick by posing as the legitimate user and downloading them from the Google Play. To ensure downloads are made only by the user, the market should require authentication before every transaction, similar to the model currently used by the iOS.

Google Play now has per-device settings allowing the user to "Require authentication for purchases" which can be set to Never, once every 30 minutes, or for every purchase. However, this is only in effect for purchases. That is, applications that are listed as "free" may still be downloaded regardless of this setting.

2.4.5 Authenticated ADB

Because of the power given through the ADB, it should not be accessible to unauthorized users. Android should require the device to be unlocked before ADB can be used. Any legitimate user should be able to unlock the device and once the connection is made, the session could be maintained by preventing the screen from locking while it is connected via USB. With ADB authentication, the attacker no longer has a backdoor to bypass the lock mechanism's authentication process, mitigating the ADB attack against obstructed devices.

Following our recommendation, ADB connections are now authenticated by default (as of Android 4.2.2, API 17). The boolean boot property `ro.adb.secure` enables this ADB functionality when set to `1`, and this property cannot be edited by the device user. The dialog depicted in Figure 2.5 appears when a USB connection is made, forcing the user to acknowledge the connection. If the device is locked, this dialog is not visible until the device is unlocked. If the user selects "Always allow from this computer" the device will not prompt again for any computer presenting the same key. The authentication is performed with standard RSA encryption (2048 bit keys) and Secure Hash Algorithm 1 (SHA1) hashing. Specifically, the protocol employs SHA1withRSA signatures. While, not particularly granular, all keys that a device has "Always allowed" can be forgotten by selecting the "Revoke USB debugging authorizations" setting (depicted in Figure 2.6). The revocation simply deletes keys from `/data/misc/adb/adb_keys`. This methodology is akin to Secure Shell (SSH) host key checking used by SSH clients to remember previously encountered

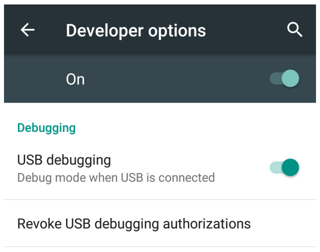


Allow USB debugging?

The computer's RSA key fingerprint is:
28:78:A5:E1:CF:1D:
67:49:A3:70:B9:BC:C8:16:2A:CD

✓ Always allow from this computer

Figure 2.5: **ADB authentication prompt**: Upon USB connection, a modal dialog requires the user to acknowledge the connection.



USB debugging
Debug mode when USB is connected

Revoke USB debugging authorizations

Figure 2.6: **ADB key revocation**: At any time, the user can revoke all of the stored ADB authorization keys.

SSH servers. The keys themselves are generated by the ADB server on the computer, typically stored in the user's home directory at `.android/adbkey` and `adbkey.pub`.

2.4.6 Trusted Platform Module

To secure a device in a managed model scenario, a root-of-trust must be established. Using a TPM provides a ground truth on which device security could be built, providing authentication of device state. Using a TPM would mitigate the recovery image attack, which relies on the ability to change the boot image. Assuming signed bytecode and authentication of the boot image, updates running unauthorized code would become extremely difficult.

While TPMs are not commonplace in today's Android devices, some TPM-like features are now present. For instance, starting with Android 4.1 (API 16), if a device was built with appropriate hardware (e.g., TI's M-Shield) the Android keystore can be backed by hardware.

2.5 Discussion

Android was designed with a focus on security, however, as new security features are added, new vulnerabilities may become available for exploitation. Recent versions of Android, in particular, Android 4.2 (API 17) and higher, have made noteworthy strides in security.

Application level permissions pose substantial complexity for users, who do not understand the permissions and therefore do not act in security-conscious ways. The complexity is not lost on developers who are not able to correctly

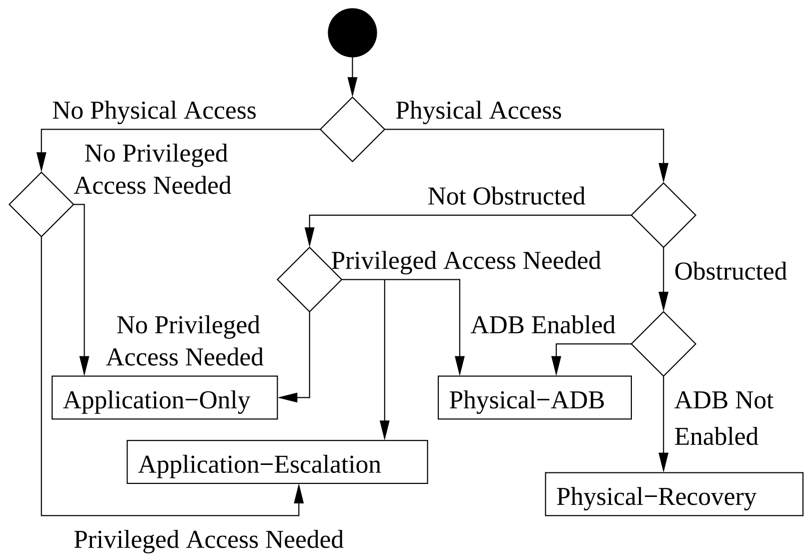


Figure 2.7: **Android attack graph**: Android attack goals and requirements; Android 4.0 (API 14) and earlier. Figure from [230].

specify sets of permissions for applications they create. In efforts to address this complexity, the permission model has been continuously revised as Android has matured.

Android's security not only depends on relationships between users and software developers, but also supply-chain relationships with device manufacturers and cellular providers. A primary security-related observation relating to supply-chain relationships is the increased latency in security patch application. Android's software publishing model adds additional steps for these entities, causing substantial delay or omission in applying patches.

This chapter builds a taxonomy of attacks against Android, and Figure 2.7 shows how an attacker could rely on this taxonomy to decide which attack path to pursue, given their own capabilities (e.g., physical access available or not, ADB available or not, etc).

We proposed six broad mitigations detailed in section 2.4. Perhaps in light of our proposals, some of these mitigations have since been employed by Google and the AOSP.

Smartphones provide many benefits to their users, but they also bring new security challenges. All the while, miscreants perpetually strive to take advantage of any lapse in security. Because of the heightened risk posed by new mobile security challenges, further steps must be taken to understand and mitigate security weaknesses.

Chapter 3

Mobile Malware Measurements and Evolution

Malicious software, or simply "malware" has certainly evolved since the days of Core Wars[1] and the now famous Morris Worm. Friendly gaming and academic curiosity have long since given way to large scale botnets [176], massive SPAM campaigns [133], cryptographic ransomware [235], and worms that traverse the globe in seconds [158]. Intentions behind malware stretch from academic proof-of-concept implementations to individual miscreants interested in quick profit to theories involving organized crime and nation-state level attackers [55].

The massive adoption of smartphones over the past several years has effected related growth in scams targeting smartphone users. Likewise, smartphones are increasingly becoming a primary computing device for many users. Today's mobile devices are used to perform financial transactions, might replace the need to carry a camera on a family vacation, and act as the primary communication hub for voice, email, test messaging, social networks, and more. The increased adoption and increased use both tend to entice miscreants aimed at leveraging this popularity for wrong-doing.

To better understand the advancement of mobile malware, we first provide insight into to historical progression of malware. In particular, we show that the immensely successful current-generation of smartphones have not only opened the door to mobile malware, but that mobile malware is following a similar evolution as found on the PC. We demonstrate this similarity, albeit on an accelerated pace, across several broad categories of malware. If the evolution continues, the broad types of upcoming mobile malware can be predicted, and related security solutions may be put in place a priori. Indeed, it now seems prudent to assume that *all* transgressions that have occurred on the PC must also be mitigated by mobile platforms. The first malware for traditional PCs took decades to appear, while the first malware for Android appeared alongside the commercial introduction of Android devices.

Evolution as a phenomenon has be studied extensively in other sciences, perhaps among the more approachable is biology. In the realm of biology, evolution is studied at various levels of abstraction (e.g., Phylum, Class, Family,

[1]A game played between two competing software programs "stalking" one another, typically until one program is terminated. The name, Core War, is derived from ferromagnetic "core memory" often employed by computers in the 1950's [89].

Genus, Species). Biological evolution also benefits from well-known terminology to represent variation between organisms, the mechanisms by which variation might occur, and the possibilities for outcome following some aspect of evolution. Indeed, evolution is so studied in biology that numerous field and laboratory studies have been and are conducted, multitudes of derivative evolutionary models have been created to explain particular types of evolution, and formal theoretical models are common [151].

When considering the evolution of malware one must not only consider the technical advancement of the malicious software, but also the intentions of the miscreants, the afflicted platforms and, to some degree, the security posture of the platforms. For example, the earliest forms of malware targeted SUN and BSD systems [95], while today Microsoft Windows platforms are targeted far more often [185]. The earliest malware for iOS only affects users that have undermined the security posture of iOS by "jailbreaking"[2] (or "rooting") the device. Whereas in Android, rooting the device is not necessarily considered a detriment to the device's security; even in Google Play one can find popular applications that only work on rooted devices. However the permissiveness of the Android platform to allow non-official marketplaces opens the platform to a wealth of malware delivered from markets that Google cannot police [225, 249].

Still in its relative infancy, mobile malware has adapted over time to evade detection and ultimately service the various nefarious purposes. Likewise, defenses are employed reciprocally, giving way to a veritable arms race.

Through measurement experiments, we evaluate to which extent existing Android markets facilitate malware installation. We build crawling mechanisms that identify a large number (195) of existing Android application markets, and gather a total of 76,480 applications from these markets, including Google Play (35,423 applications). From this application corpus, we show that *application repackaging*, in which miscreants disseminate malware posing as legitimate, well-known applications, presents a noteworthy threat. By analyzing signing strategies used in alternative marketplaces, we show that some malicious markets extensively reuse certificates to provide valid signatures on maliciously repackaged applications.[3]

In this chapter we briefly chronicle the origins of malware in section 3.1, including the advent of and differences embodied by mobile malware (section 3.2). In section 3.3, we provide malware measurements sampled from existing mobile software marketplaces. A mapping of mobile malware to a specific type of evolution: the Evolutionary Arms Race [88] is provided in section 3.4. In section 3.4.1 we link mobile malware evolution broadly to that of PC malware in a study of *macroevolution*. Complimentary to the broad analysis, in section 3.4.2 we provide a detailed analysis of related malware specimens demonstrating malware *microevolution* within a specific attacker campaigns and malware families. We consider related work in section 3.5 and conclude the chapter with discussion in section 3.6

[2]The term *jailbreaking* refers to the action of removing typical administrative limitations on modern smartphones, often by circumventing security controls put in place by the operating system authors—a privilege escalation attack. In the context of smartphones, the term is synonymous with *rooting*.

[3] Portions of this chapter previously appeared in [225].

3.1 Background

Compared to traditional PCs, mobile phones are technologically progressing much faster, effectively "catching up" with traditional PCs. With today's smartphones, parallels can be made solely based on hardware with recent model phones having comparable processing power as a laptop only a few years old. Likewise, mobile malware first appeared about 15 years after the commercial introduction of first-generation smartphones. As with the technological progression, malware targeting smartphones is effectively catching up with traditional PC malware. In order to frame discussion on the increasing prevalence and sophistication of mobile malware, we provide a brief background of pertinent advancement of malware.

3.1.1 Origins of Malware

The programmable computer dates back to 1936 with the Z1, or systems like Colossus, ABC, or ENIAC in the early 1940's. However, with the early incarnations of the Internet in the 1960's and wider adoption of the more affordable PCs in the 1980's, computers became increasingly connected and therefore much more accessible to remote parties.

For completeness, Von Neumann's work on self replicating automata from 1949 must be mentioned [166]. However, more concretely, the notion of creating "offensive software" might be traced back to Core Wars in 1984. In these battles, programmers would create software designed to be pitted against another programmers' creation. Each program would be executed one instruction at a time until one program caused the other to terminate (or some other property was violated, such as the exhaustion of a memory limit) [89]. While Core Wars is colloquially used as an early example of such "offensive software," the concept can actually be traced back even further to 1961, where a small group on engineers created a game known as Darwin [101]. The core concept of Darwin is similar to Core Wars, but only three people reportedly ever participated in Darwin's few-week lifespan. One of these programmers was Robert Morris who effectively permanently ended the game by creating an unbeatable program.

The origins of malware are often traced back to another of Morris' creations, the Morris Worm [95], in 1988. This worm was reportedly an attempt to measure the size of the Internet at the time, but the unintended side effects of the software exploits targeting VAX, BSD and SUN machines were catastrophic. The monetary damage estimates and general realization about the fragility of the Internet prompted the Defense Advanced Research Projects Agency (DARPA) to create the first Computer Emergency Response Team (CERT).

So, early forms of offensive software were likely created some 20 years after computers became generally programmable. The first such network-oriented software was release 10-15 years after the Internet started gaining wide adoption.

In the decades following these provoking thought exercises, PC malware has proliferated to a scale that brings computer and network security to the front page of the popular press as well as regular, closed-door discussions of national security. Entire multi-billion dollar industries have been formed around securing computing devices and protecting data. For example, so-called AV software aimed to mitigate a variety of malware can regularly be found on consumer computers and enterprise workstations alike.

3.1.2 Origins of Mobile Malware

Of course, the history of the telephone can be traced though many related technologies arguably originating with Alexander Graham Bell's 1876 patent or indeed the telegraph in the early 1800's [120]. The origin of the mobile telephone is easier to pinpoint with Motorola's 1973 introduction of a portable (4.4 pounds) telephone [85]. Portable telephone's became commercially available in the mid 1980's. Features taken for granted now were developed in the coming years adding complexity to devices and networks. For example, SMS first became available in 1992.

The network evolution, and the services it could support are mirrored with devices capable of taking advantage of these services. At a high level, First-Generation (1G) services were analog and meant to support voice communication. 2G was aimed to support voice communication but with digital communications. Using the same 2G standards simple data services were added constituting what is often called 2.5G. 3G was designed with data service in mind (in addition to voice), and 4G was designed primarily for data services and speeds comparable to land-based broadband connectivity.

The concept of a smartphone dates back to 1971 [174], but devices were not commercially available until the mid-1990s and were not really popular in the US until the early 2000's. Symbian released the first Symbian OS device in June 2001, and the OS quickly gained dominate global market share which was maintained until late 2010. There are other notable technological milestones, but today, mobile operating systems have matured and term "smartphone" typically refers to devices like the iPhone, those running Android OS, or those running something else with similar capabilities. All other less-capable mobile telephones are now referred to as "feature phones."

Therefore, the origin of current generation of smartphone technology is often credited to Apple with the 2007 introduction of the first iPhone. Android's first device, the HTC Dream, reached markets in October 2008. WebOS, Windows Mobile, Windows Phone, Blackberry and others continue to be part of the ecosystem, but Android and Apple (iOS) devices easily dominate most markets today. These platforms are much more mature and modular than feature phones. Feature phones are often loaded with one particular software set, sometimes called firmware, that is seldom altered or updated throughout the life of the device. Conversely, smartphones are based on conventional operating systems (BSD/OSX, Linux, and Windows) which can be more easily updated and the functionality of a smartphone can be extended beyond its base capabilities by download mobile applications (or simply "apps").

Much like the situation with traditional PCs, early portable telephones did not have much issue with malware. Like early computers, early portable telephones had very limited computational power and connectivity. Generationally, mobile telephones and cellular networks became more powerful and the features available to users richened. In 2004, malware started to appear, primarily targeting Symbian OS. As was the case with traditional computers, early forms had less malicious intent or were fostered from curiosity. For instance, in June 2004, Cabir malware was created as a toy example to simply propagate via Bluetooth. Also in 2004, a game called Mosquito sent SMS messages to the game vendor without user consent, sometimes incurring cost to the Symbian user. Quickly following these somewhat innocuous examples, was Pbstealer in 2005. Pbstealer, is Symbian OS malware designed to steal address book information, and like Cabir, transmit via Bluetooth.

In 2005, CommWarrior became the first malware to employ a truly mobile-oriented feature: Multimedia Messaging Service (MMS), a 2G data service. While the implementations vary drastically, MMS is often thought of as an extension of SMS facilitating data communication beyond the SMS limitation of 160 characters. CommWarrior would spread via Bluetooth and via MMS, automatically sending copies of itself to entries from an infected device's address book. Like Mosquito, CommWarrior could cause financial harm to the victim due to amassing MMS charges.

Even though malware like CommWarrior and Mosquito had potential to cause financial harm to victims, the harm was via incurring normal cellular-provider charges for unintended transmissions. In this model, the victim has financial loss, but the attacker does not have direct financial gain. In 2006 this changes with RedBrowser Java malware, which was coded to send text messages, one after another, to a premium SMS number. By using a premium SMS number, the SMS equivalent to calling a toll 1-900 number, the attacker could now realize direct profit, while the victim experienced greater loss. This malware afflicted mobile devices that made use of Java 2 Micro Edition and therefore could infect device across different platforms.

The first Android malware was created in September 2008, again born from curiosity. Malware created as an example was able to disable all communications in and out of an Android device, steal information, accept all calls, etc (Radiocutter, SilentMutter, CallAccepter & StiffWeather) [73]. The first Android malware with truly nefarious goals was discovered in August 2010. This fake application misled victims into installing an application then then sent messages to premium SMS numbers, like RedBrowser, causing victims to pay for these SMS messages and the miscreants to profit.

In the following years, mobile malware has advanced to sophisticated attack campaigns. As in PC malware, intent varies, some actors may be politically motivated, while others may be financially motivated. Also bearing similarity to PC malware, some malware may be multi-stage, multi-platform, resist analysis, and so on. In a broad classification, some types of attacks resemble those found on PCs. For instance, so called "ransomware" locking users out of their own devices, or actor-controlled sets of infected devices—botnets. Other core features of malware, such as propagation, commonly use tactics other than those prevailing in PC malware.

3.2 Why Mobile Differences Matter

Today's mobile device market is fickle. Worldwide adoption rates of smartphones are growing rapidly, and the prominent software platforms of choice are clearly Apple's iOS and Android. However, cellular carriers strive to bring some market differentiation to pull customers from other carriers and keep their existing customer base. Likewise, manufacturers seek to sell more product, meaning the devices they create must either reach new audiences or the existing consumers must acquire more devices. Of course, updated hardware theoretically means greater capability, the ability to run more complex software faster, and the ability to communicate with newer generations of cellular network.

In some cases a consumer may seek a device upgrade to acquire hardware (e.g., a better camera), but today upgrades are often sought for other reasons. For instance, US carriers often subsidize the cost of a new device, and in return the consumer is contractually obligated to maintain a monthly plan for a set term (e.g., 24 months). However, consumers may, and are encouraged by the carrier, upgrade a few months early in exchange for commitment to a new 24-month contract. Or, consumers might consciously decided to pay an unsubsidized price for every other device, resulting in a new device about once per year. In 2007, US consumers replaced mobile devices in average of just over 18 months. In 2010, this rose to nearly 22 months [102], and to 26 months in 2014 [103]. The increased duration might be attributed to decreased subsidy, lack of compelling features, or perception that the incumbent devices still offer sufficient functionality, or otherwise. However, even at 26 months, mobile device replacement cycles are much shorter than that found with other consumer electronics.

Take into account that the base software, often simply referred to as the "OS," is released much more often, every 16 weeks for Android (average API version 1 to 21) or every 11 months for iOS (average version from 1 to 8). Each of these updates bring relatively large changes to the operations of the devices, often changing User Interfaces (UIs) drastically. These OS updates are now delivered almost exclusively Over-the-Air (OTA) meaning that the devices retrieve the updated software automatically requiring little interaction from the user. However, as detailed in section 2.2, these updates are typically only available for relatively recent devices—a physical device may only be able to receive one or two major updates before it is no longer supported by the current software version. This presents another reason for consumers to upgrade to a newer device, but a side effect of this phenomenon is that the outgoing devices are those that will quite literally never receive another official update. This is in sharp contrast to traditional computers that enjoy long OS lifespans and use in secondary markets running different, less resource intensive software. Figure 3.1 shows the major versions of iOS and Android base software updates. There might be more frequent "minor" updates, often for security patches. However, due to administrative limitations, getting these updates applied to devices in a reasonable time frame is often an issue, especially on Android.

Mobile devices administrative models have been explored often in other works [230], and in summary, the models are very restrictive compared to typical computers (see also section 2.2.1). PC administrators often are able to install

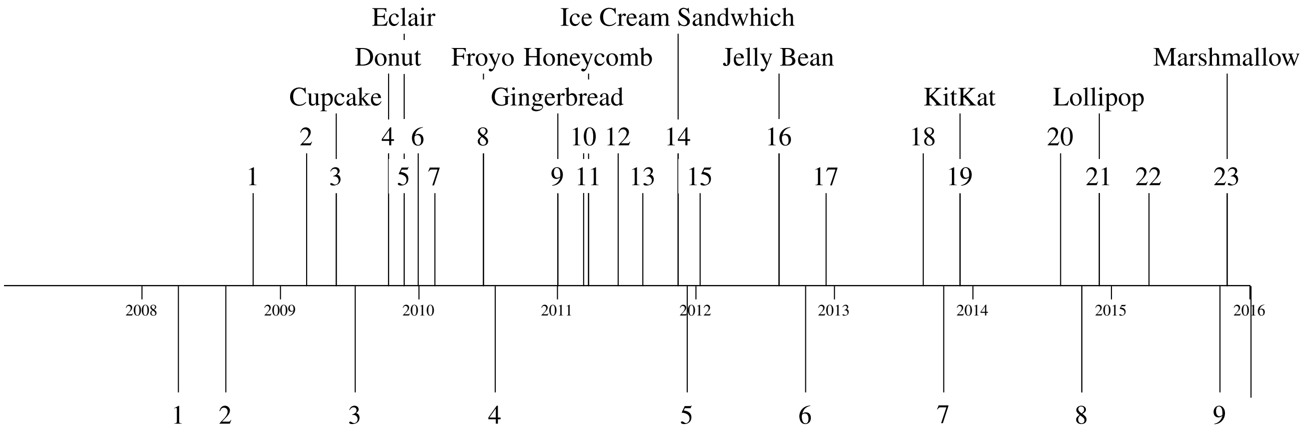


Figure 3.1: **Android and iOS chronology**: Android (above timeline) and iOS (below timeline) API versions over time. The average time between an Android API update is 112 days—a much shorter time frame than when consumers replace smartphone hardware. Even when considering the “named” versions of Android (e.g., Eclair, Jelly Bean, or Lollipop) the average only extends to 223 days. In iOS the average time between major version updates is 341 days

software “from scratch,” bringing a PC only having simple Basic Input/Output System (BIOS) and firmware to a fully usable machine. In this way a PC can be reset to a known state by repeating this process, reinstalling an operating system, thereby logically erasing the previous state of software on the machine. This is effectively not possible on today’s smartphones, where there is no interface in which to “load an install Compact Disc (CD)” or similar. Moreover, the software is often tightly controlled with cryptographic signatures and protected boot loaders precisely meant to discourage wholesale replacement of device software. This same administrative model restricts the ability to deploy traditional protections found on PCs. In fact, there is really no clear model for any third party to add any pro-active protections to a device nor any mechanism to fully remediate a device—that is restore to a known state. With today’s software administration models the only entities really administering a device are the base OS software maintainers, and in some cases the manufacturers and cellular providers.

Mobile platforms also give way to new methods of infection and propagation. In fact, early mobile malware often propagate via wireless peer connectivity offered by Bluetooth. More complex networks and devices have offered ever more connectivity (though thankfully, Bluetooth connections now typically require authentication). Mobile malware has many options for propagation, including Bluetooth, MMS, WIFI, removable media, Near-Field Communication (NFC), and “push services” that routinely communicate with devices and connected computers. When considering infection, new models of interaction must be considered as culture continues to change in favor or mobile-oriented interactions, such as paying for goods by "touching" a device to a point-of-sale [70] or photographing two-dimensional barcodes to gain information about a nearby sign [228]. Common infection techniques on the PC, such as "drive-by-downloads," have not yet been fully realized in mobile malware.

Today, mobile malware is typically disseminated via unofficial software marketplaces [225,249]. Such markets are officially supported by Android, though Google maintains the official marketplace, Google Play, and strives to have users prefer to use this market over others. iOS does not officially support secondary markets, so an iOS device user that wishes to use a marketplace other than Apple's App Store must jailbreak the device. These Unofficial markets might serve legitimate purposes, such as offering applications not permitted in the official marketplace for one reason or another, or indeed offering a marketplace in geographic locations in which the official market is not permitted to operate. Unofficial markets might also serve no legitimate purpose, existing solely to disseminate malware [225]. Mobile malware is often found to be localized to a particular region which may be due to a pre-disposition due to the localization of software distribution channels.

3.3 Measuring the Prevalence of Malware in Online Marketplaces

Each market may have different policies for policing applications. Google maintains a reactive policy in the official Android market, but alternative markets may have a less effective policing policy or no policy at all. To demonstrate the threat of application repackaging we investigated the presence of repackaged applications in existing markets. In this section we discuss our measurement methodology to create a corpus of alternative market applications, how we created a corpus of official market applications, and a description of the resulting corpora.

3.3.1 Collecting applications in alternative marketplaces

In order to create a corpus of applications from alternative markets, we first conducted an experiment to observe alternative distribution mechanisms.

We identified 194 alternative marketplaces by popularity based on search engine results. First we created a list of candidate site seeded with search results for "alternative market android," "third party android market," "free android applications," "android app store," and simply "android market." We then expanded the list of candidates to include the same strings translated to all 63 languages currently supported by Google Translate. We manually inspected search results to prune candidate sites that did not actually deliver mobile applications (many only offer meta-data, directing an interested user to then download the application from the official market). During manual inspection of search results, we appended obvious links to other marketplaces to the list of markets. Perhaps the most important observation from the search results inspection is that, unlike Google Play, applications from some alternative markets can currently be downloaded using common, unauthenticated Hypertext Transfer Protocol (HTTP) methods. In many cases the Uniform Resource Locator (URL) for applications is highly predictable and can facilitate complete coverage, as in `http://yadroid.com/?download=n` where `n` is between 1 and 2696.

We then used the list of market candidates as input to a web crawler to automatically download sets of mobile applications. We configured the crawler to further expand the set of domains by following links to other domains.

We observed that naively following references to other domains resulted in downloading large amounts of data from sites that were obviously not alternative markets, such as `sourceforge.net`. Similarly, during the collection pilot we inadvertently obtained several gigabytes of fonts, music, feature films, pirated PC software,[4] for instance. Our first attempt to limit results to Android applications was to limit downloads to files with an ".apk" file extension. Filtering by file extension proved to be too limiting. By comparing a dataset obtained without the file extension filter to a dataset obtained using the filter, it was obvious that applications from entire sites were excluded. Through this iterative process we identified several delivery methods that must be taken into account when crawling current alternative markets.

While an Android application is a Zip archive packaged in a certain way (see section 6.1), there is no guarantee that a given marketplace delivers applications to the device in this form. In fact, we observed many other methods during our collection pilot. For example, one site delivers applications as expected, but the file extension is ".ipa" instead of ".apk." Similarly other sites also deliver the application archive as expected but with no file extension all, and are instead accessed by a URL such as `http://yadroid.com/?download=260`. Yet others will "double package" the application for delivery, resulting in a Zip (or other) archive that contains a Zip file that is an application. For these reasons the crawler was altered to be very permissive resulting in downloading content that is not useful for this study.

Instead of automatically deleting very large files ($>$ 200MB) we conservatively opted for manual inspection. Nearly all files of this size proved to be innocuous, such `.iso` files of Linux distributions, yet some sets of applications were found to exist in large archives. We decompressed the archives to make accessible the applications found as files in archives. By automating the process of recursively uncompressing gzip, rar, and zip files (identified using the Linux `file` utility), we extracted mobile applications from both the large archives and smaller "double packaged" applications.

After recursive decompression, we used the Linux `file` utility to positively identify files that were not mobile applications, and we subsequently deleted these files. In Table 3.1, we show the list of files types that we observed in an unfiltered dataset after running the crawler. We constructed the list by appending types to this list iteratively as we observed new, undesired file types in the downloaded dataset.

[4] Many of which contained PC targeted malware.

Microsoft PE Executables
XML Documents
Image types (e.g., PNG, JPG, GIF)
Office formats (e.g., Microsoft Word)
empty files
text files (e.g., ASCII, ISO-8859 C program)
PDF documents
media files (e.g., MPEG, MP3)
fonts (e.g., TrueType font)

Table 3.1: **File types pruned in post-processing** Files of these types were downloaded by the crawler and subsequently filtered.

After the coarse-grained pruning based on file types found in Table 3.1, we tested each file to observe if the file was a ZIP file that contained the `AndroidManifest.xml` binary Extensible Markup Language (XML) file. If so, we classified it as a valid Android application.

We used the above described collection and pruning process to collect 41,057 applications from 194 alternative markets in October 2011. The identification of markets and subsequent downloading of applications is biased by popularity, both by using search results to identify marketplaces, and by the likelihood that popular applications are made easier to locate by each marketplace interface.

3.3.2 Collecting applications from the official Android market

Google Play is intended to only be accessed directly from Android devices. Even when "installing" an application using the online interface at `play.google.com`, a signal is pushed directly to a device associated with user authenticated in the browser session.

Even though the Android framework is open source, many software components found on Android devices are proprietary, including the Google Play software. We created a protocol-compatible tool that facilitates granular access to applications in the official market. We designed the tool iteratively by reverse-engineering the market components found on an Android device, observing network traffic during a market transaction, and by inspecting server responses after manually adjusting protocol parameters.

The official market protocol requires authentication and a device identifier. To authenticate to the service, we created a new user using an actual smartphone, and extracted the device identifier from the device. The username, password, device identifier, and SDK version are used to establish a market session. This session is similar to the session status that would occur when opening the *Play Store*[5] application, the official client, on an Android device.

Once a session is established the server is queried for a list of application categories (e.g., Finance, Education, Medical). Much like the official client, server results vary depending on several parameters. By manipulating these

[5]Or *Market* application on older devices.

parameters, a client can obtain different results to mimic views present in the official client such as "Featured" or "Top Free" in any category. For example, applications can be ordered by any of POPULAR, NEWEST, FEATURED, NONE; or the field can be omitted.

In addition to mimicking the capabilities of an official client, we are able to manipulate additional parameters, such as the wireless carrier associated with this client. A physical device is typically associated with a single carrier and the official client simply utilizes the carrier associated with the device. We can enumerate sets of carriers impersonating devices on several networks by altering the Mobile Country Code (MCC) and Mobile Network Code (MNC) as defined in ITU E.212. For example the United States has MCC's 310-316, and MNC 260 specifies T-Mobile. We can iterate 310012, 310410, 310120, and 310260 to impersonate devices from Verizon, AT&T, Sprint and T-Mobile, respectively.

The ability to impersonate devices from various networks is important for coverage as some applications may only be made available to customers on certain networks. Likewise, certain applications only exist for certain types of device hardware, geographic region, and software versions of Android. As of October 2011, any particular combination of market parameters would return a maximum of 800 results, due to this limitation, it is not possible to simply iteratively collect all applications in the entire market.

The actual applications are not downloaded through the existing market session. Market query results contain meta-data about applications. Some of this meta-data is available in the official client such as the title, cost, and review ratings. Other meta-data is not visually displayed, such as the application's AssetID. The AssetID is needed to download applications independent of the existing market session. The AssetID is approximately 20 ASCII digits long and must be precisely specified in order to download the application.

When attempting to download too many applications over a period of time, the server may not permit further downloads, temporarily blocking connections. We observed that HTTP 503 errors precede such blacklisting, and accordingly implemented a back-off procedure.

To create our corpus, we collected free applications from each application category as accessible from the United States on four carriers: Verizon, AT&T, Sprint and T-Mobile. We additionally iterated through known Android and SDK versions, eventually collecting 35,423 applications in October 2011. Because we collected by category, the 35,423 collected applications are biased by popularity in each category. Furthermore, due to the complexity of automating the payment protocol, we only collected free applications.

3.3.3 Results

We next discuss the prevalence of malware in the marketplaces we have measured. From our collected corpora, we identify known malware attributable to each marketplace. We also offer particular measurements (e.g., on certificate reuse) from the corpus to inform discussion of the protocol we later present in section 6.2.

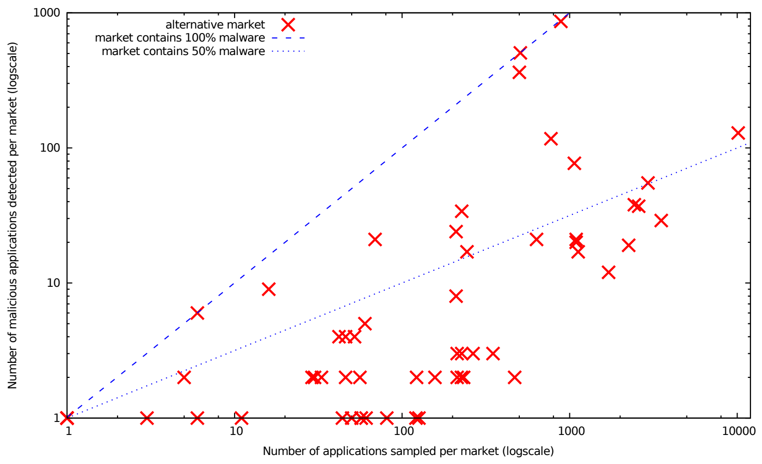


Figure 3.2: **Alternative market malware**: Total applications and detected malware per market. Each point corresponds to one measured market. Points higher on the Y-axis delivery higher percentages of malware. Points approaching the dashed line deliver malware exclusively. Points near the dotted line deliver malware 50% of the time. Figure adapted from [225].

To determine a lower bound on malware present in each market, we scanned each file with multiple AV products, through the VirusTotal API [233].[6] VirusTotal is a service that offers file scanning through many different AV products by vendors such as Symantec, McAfee, Kaspersky, and TrendMicro (42 during our measurements). Despite the large number of AV products being used, malware detection in this manner remains a very conservative lower bound as mobile malware detection is less mature than desktop malware detection. Some reports show that many mobile AV detection rates are very low: between 0% and 32% [180]. It is highly unlikely that any so-called "zero-day" malware would ever be detected under this procedure. Therefore, the actual delivery of malware from marketplaces is quite likely a larger problem than we show.

Some alternative markets appear to be completely absent of malware, but a few markets distribute malware almost exclusively. In the scatterplot shown in Figure 3.2, each market we crawled corresponds to one point, whose x-coordinate denotes the total number of applications in that market, and whose y-coordinate denotes the number of applications detected as malicious in this market. The dashed line in Figure 3.2 represents the threshold where *every* application sampled from a market would be detected as malicious (and hence, no point can be above that line). Several points approach this line, demonstrating that our naive sampling identified a number of markets which almost exclusively distribute malicious applications. Particularly preoccupying is the case of the markets in the top right corner of the graph. Not only do these markets have very high percentage of infected applications, but they also provide a large number of applications.

We can further use this data to attempt to exhaustively classify all Android malware as repackaged or some other type of malware. After eliminating "potentially unwanted programs" detected by AV, such as spyware, we can con-

[6]Due to limitations to the VirusTotal API, applications larger than 20 MB were not scanned.

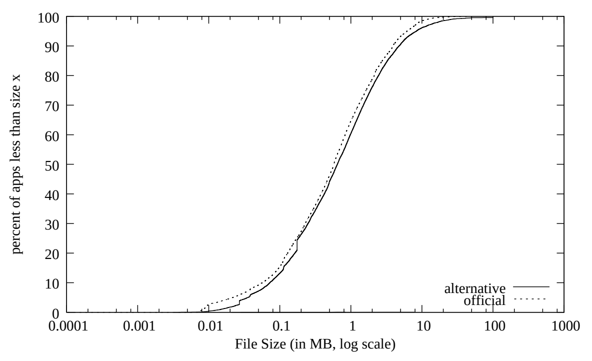


Figure 3.3: **CDF of market application sizes**. Alternative markets are slightly larger than official marketplace applications. More than 40% of applications are greater than 1 MB. Figure from [225].

cisely catalog all known malware as of November 2011. We observed 55 different families of malware, 40 of which (or 73%) employ some type of repackaging or spoofing (such as those enumerated in [109] and more exhaustively in [248]). We note that many early Android malware families as well as the most recent employ some type of repackaging.

As another datapoint, other researchers use a combination of package name comparison to applications found in the official market and manual analysis to classify repackaged applications [248]. They similarly find that 86% of unique samples in their Android malware corpus are repackaged. The reported number is different from our observation for two reasons: First, their described corpus is entirely comprised of malware samples, many of which belong to the same family leading to a non-uniform distribution across malware families. Second, the definition of *repackaged* is slightly different and does not include the set of applications we term *spoofed* in section 6.1.2.

We next look at possible indicators of malware distribution strategies. We first measured the number of applications which provide package names that form valid domains. To do so, we parse each package name with the Perl module `Data::Validate::Domain`. We find that 83% percent of the applications originating from the official marketplace have package names that, when reversed, represent a well-formed domain, following Google's suggestions (see section 6.1) for package naming conventions. Interestingly, applications from alternative markets exhibit a slightly higher rate at 86%. This seems to indicate that exotic naming conventions are not indicative of malware. On the contrary, we found that, in markets with the highest percentage of malicious applications, applications tend to comply *more* with the proposed standard naming conventions. In hindsight, this does not come as a surprise: malicious applications designers have incentives to make their applications "blend in" as much as possible.

Figure 3.3 shows the distribution of application sizes for the official and alternative Android markets. Approximately 40% of all applications are greater than 1 MB. Alternative market applications are generally slightly larger than those in the official market, but the size distribution between the market types is clearly similar.

Finally, we attempt to characterize which signing strategies are being used in alternative marketplaces. The lack

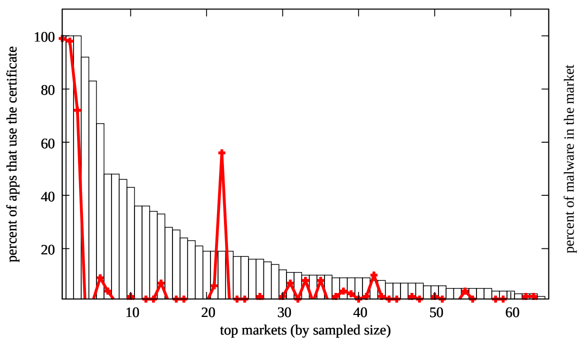


Figure 3.4: **Certificate reuse in alternative markets**: Percentage of apps which share the same certificate (bars) overlaid with the percentage of malware (line plot). Each point along the x-axis corresponds to one measured market. Markets are ranked by decreasing reuse of certificates. Only the top 64 markets (in number of applications) are presented. Figure from [225].

of a Public Key Infrastructure (PKI) and general lack of proper certificate validation does not encourage adoption of best practices. Indeed the near-ubiquitous use of self-signing certificates enables the publisher to adopt a number of different signing strategies. For instance a publisher could use the same certificate to sign every application they publish, or could use a different certificate for each version of each application. Shortly stated, certificates do not provide any guarantee on the application integrity, or origin, and patterns of certificate misuse may be evidence of application repackaging.

We observed heavy re-use of signing certifications in our collection. Indeed, only 52% of certificates from the official market were unique. The distribution of certificates is not uniform, as some certificates are used as many as 693 times, while others are only used once. In alternative markets, the signing strategies vary drastically. Some markets exhibit distributions similar to the official market, while others use a single signing certificate.

Figure 3.4 plots, for each market among the 64 markets that distribute the most applications, the highest percentage of applications in the market that are signed using the same signing certificate. We overlay the plot with a line showing the corresponding percentage of malware in the same markets. Strikingly, almost all the alternate marketplaces with high percentage of malware appear to substantially re-use signing certificates. Across all measured markets there is a strong linear correlation between the percentage of malware and the percentage of certificate reuse (Pearson's $\rho \approx 0.64$). In the seediest markets with close to 100% known malware, *all* applications are signed using the same signing certificate. In this case, the remaining applications to make up the 100% are quite likely malware that is not yet detected by AV.

In sum, these results provide clear evidence that malware in alternative markets is a problem we cannot neglect. As a point of comparison, in the official market, we discovered 119 applications containing malware, or 0.003% of all applications we surveyed. While certainly very low, this number needs to be taken with caution: these are the

Malware	Discovery	Platform	Significance
Cabir	2004-06	Symbian OS	Widely regarded as the first mobile malware
Mosquito	2004-06	Symbian OS	First to cause direct financial harm (via incurred SMS costs)
CommWarrior	2005-03	Symbian OS	Implements MMS propagation; first real propagation threat
CardTrap	2005-09	Symbian OS	Attempts to infect PCs via removable media; renders mobile device unusable
RedBrowser	2006-02	J2ME	Profit model using Premium SMS; cross-platform
AppSnapp	2007-10	iOS	First jailbreak for iOS
StiffWeather	2008-08	Android	First Android Malware (one of a set including CallAccepter, Radiocutter and SilentMutter)
Yxes	2009-02	Symbian OS	ability to secretly send SMS and create Internet connections (to C2 server)
ikee	2009-11	iOS	First iOS malware, a worm that afflicts jailbroken devices
asroot	2009-08	Android	First jailbreak for Android
Toires	2009-11	iOS	Capable of information theft without violating iOS security properties; jailbreak not required
Zitmo	2010-09	Symbian	Mobile malware created to work in concert with PC malware
LBTM	2010-09	iOS	Financially-motivated malware distributed in Apple App Store; jailbreak not required
Geinimi	2010-12	Android	First mobile botnet allowing for attacker remote control of a device
Dogowars	2011-08	Android	First public protest malware meant to combat controversial app Dog Wars
Gingermaster	2011-05	Android	Utilized a root exploit granting greater control of the device than the user has; also A/V evasion
Chuli	2013-03	Android	First politically-motivated malware, targeting human rights activists in Tibet.
FakeDefend	2013-06	Android	First aggressive ransomware locking the device.
svpeng	2013-07	Android	Obtains "Device Admin" capability found in the Android API 8
FakeInst	2013-08	Android	While FakeInst is not new, this is the first time Google Cloud Messaging is used for C2 (among others)

Table 3.2: **Notable milestones in the evolution of mobile malware (2004-2014).** Malware names, discovery, platform and significance for notable mobile malware specimens.

applications that were detected by AVs as being malicious, and are therefore a strict lower bound on the total amount of malware actually present in Google Play.

3.4 Mobile Malware Evolution

With measurements of the mobile malware problem space in hand (section 3.3), we turn to analyzing the evolution of mobile malware. Much as biology studies evolution at different levels of an established taxonomic ranking, it makes sense to evaluate mobile malware at different levels of abstraction. Without a formal taxonomy, we simply turn our investigation to macro and micro analysis of mobile malware evolution.

Seminal work also considers inter- and intra-species arms races [88]. The lack of a concrete taxonomy for malware, makes such comparisons difficult. If "species" is scoped to mobile malware, then some comparisons later in this chapter may be considered inter-species. However, if the "species" is software or malware, then the later comparisons are intra-species. Solely to establish terminology for the remainder of this section, we adopt "species" to mean broad groupings of malware afflicting a culturally accepted computing paradigm. Specifically, we consider two species of malware: mobile malware and PC malware.

3.4.1 Macroevolution of Mobile Malware

When analyzing mobile malware as a whole, common features emerge facilitating historical hindsight in how the malware has evolved generally. Table 3.2 enumerates notables milestones in the history of mobile malware. Each denotes the first occurrence of a class of malware or the first occurrence of a notable technique. For example, FakeInst as a family has existed for years, but in 2013 the malware changed to use Google Cloud Messaging (GCM) in lieu of a standard Command and Control (C2) infrastructure. Targeting various platforms, the malware specimens found in Table 3.2 represent innovation in areas such as communications (Bluetooth, MMS, GCM), privilege escalation (rooting, device admin[7]), and motivation (political, ransomware, financial). While it is impossible to strike the balance between innovation of mobile malware and the completeness of including every possible specimen in the table, Table 3.2 clearly demarks discrete progression of malware from simple toy programs, to profit models, to cloud service C2. However, macro evolution can be quantified with other intra-species metrics such as growth and technological advancement, or comparisons with other malware—a type of inter-species analysis.

Intra-species: Growth and distribution

Today, most mobile malware clearly targets the Android platform [73, 185]. This is true when considering raw permutations (usually identified by unique cryptographic hash) or some slightly more meaningful grouping, such as number of unique malware families. The trends clearly indicate that Android is the platform of choice by those developing and deploying malware.

For example, Figure 3.5 depicts the "mobile malware market share" for the past ten years (data collected, and averaged from industry reports [52, 83, 119, 136, 146, 152, 192, 218, 219]). In the infancy of mobile malware, malware targeting Symbian was more prevalent. However, in recent years, Android malware has clearly dominated the plot. Of course, taking into account that Android did not even exist until late 2008, and that Symbian's popularity has sharply decreased, popularity of the platform is likely a factor considered by miscreants plotting attacks.

The relative lack of malware for Apple's iOS should be taken with a small grain of salt as reporting is often skewed in that the determination of malicious behavior is often based not only technology but also policy. For example, consider an application that contains software subroutines designed to collect address book information from a device. Such behavior may be cause to mark an Android application malicious, yet be deemed as acceptable behavior for applications in Apple's App Store. Examples of such policy decisions are easy to point out with Aurora Feint which collected the entire address book (eventually removed from the App Store in July 2008) [77], Storm8 was sued for collecting users' phone numbers [65], or Path social media application that collected address book and other sensitive information in 2012 [179].

[7]Introduced in the Android API 8 this slightly elevated permission for software grants access to a few additional APIs and can make software granted this permission difficult to uninstall.

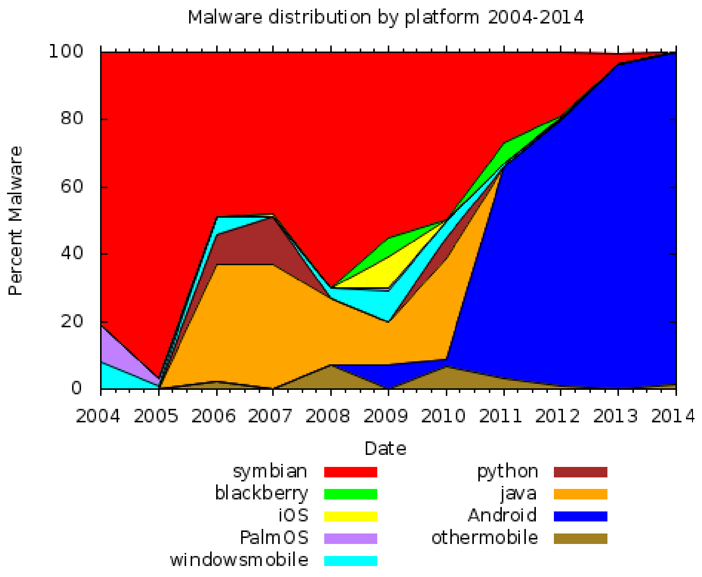


Figure 3.5: **Mobile malware "market share"**: Symbian, the most popular mobile phone manufacturer in the mid-2000's, initially accounted for the largest percentage of mobile malware. Now, Android, currently the most popular smartphone platform, easily accounts for most mobile malware.

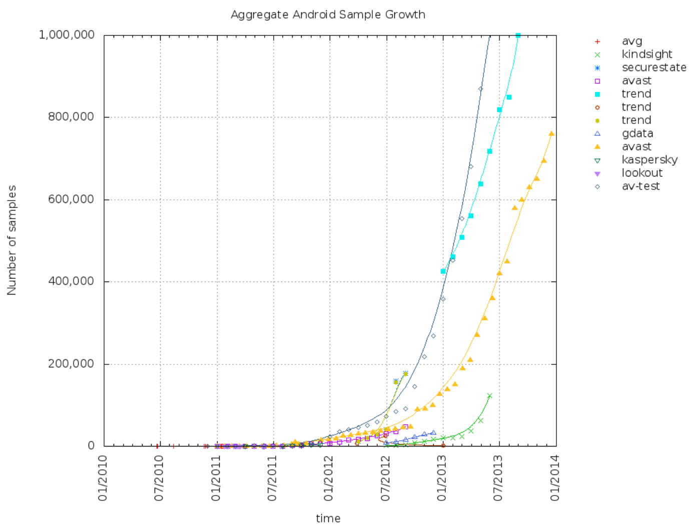


Figure 3.6: **Aggregate plot of unique Android malware specimens**: Each industry report indicated exponential growth. While collection and reporting techniques vary for each dataset, the trend is clear.

It is important to note that Figure 3.5 is plotted as a percentage of malware specimens. A percentage plot is required due to the massive growth in malware. Figure 3.6 shows an aggregate plot of Android malware measurements as reported by numerous security industry sources [60,79,91,105,115,119,129,152–155,165,193,214]. Each dataset shows the exponential growth over some reported range during the relatively short lifespan of Android, ultimately showing a handful in 2010 to approximately 1 million in 2013. Of course, collection statistics, such as those reported in Figure 3.6 are reported cumulatively (it is not possible for a line to trend down). However, the exponential trend shows undeniable growth, not stagnation. Growth charts can be similarly constructed for other platforms such as iOS or Windows Mobile, however the scale is substantially smaller than that in Figure 3.6. Even with the "market share" clearly dominated by Android, it is clear that mobile malware is a prominent, rapidly growing threat.

Intra-species: Systemic technological advancements

Malware authors often seek to hinder detection and analysis of malware. One mechanism for such hindrance is *obfuscation*—that is attempting to make software unintelligible during analysis. Perhaps the most simple form of obfuscation entails simply encoding data. For example, base64 encoding was used by SmSBoxer in 2009. Base64 decoding was also used in a novel way in 2011 in that after the decode, every other character was used to decode URLs.

Another simple technique is Exclusive OR (XOR) encryption (often also denoted as obfuscation). In XOR encryption, the bit-wise exclusive logical OR operator is used to combine a key with the data to be encrypted. A strong XOR requires the length of a random key to be equal to the length of the original data, resulting in encrypted data indistinguishable from random. However, in most malware, very short keys are employed, often only one byte, lending to the tendency to think of XOR as obfuscation rather than encryption. XOR encryption is used frequently in malware, including WinCE Pmcryptic in 2008, SymbianOS Yxes in 2009, SymbianOS ShadowSrv in 2010, and Android DroidDream in 2011.

Of course, relatively robust *encryption* is also employed by malware, and evolution can be observed here as well. Looking at early Android specimens employing encryption from 2011, Data Encryption Standard (DES) encryption can be found in Geinimi (Jan), Hongtoutou (Feb), and DroidLight (June). However, the relatively weak DES promptly gives way to the stronger AES with DroidKungFu (June) and later NotCompatible (May 2012).

Inter-species: PC malware growth

Studies of biological evolution indicate that growth rates might vary substantially across major taxonomic ranks [88, 206, 208]. Again, the comparison suffers from lack of terminology, but for discussion purposes let us consider PC malware and mobile malware to be peer species found at the same taxonomic rank.

Malware growth observations similar to section 3.4.1 can be made about PC malware. Namely, that the most dominant platform, Microsoft Windows, also is afflicted by the most malware. While Apple OSX and Linux malware exist, and demonstrate growing graphs of similar shape, the scale of malware is much smaller on these lesser deployed platforms. Figure 3.7 depicts aggregate PC malware quantity over time compared to mobile malware (aggregate of points from Figure 3.6). When considered in relation to mobile malware, PC malware currently far exceeds mobile

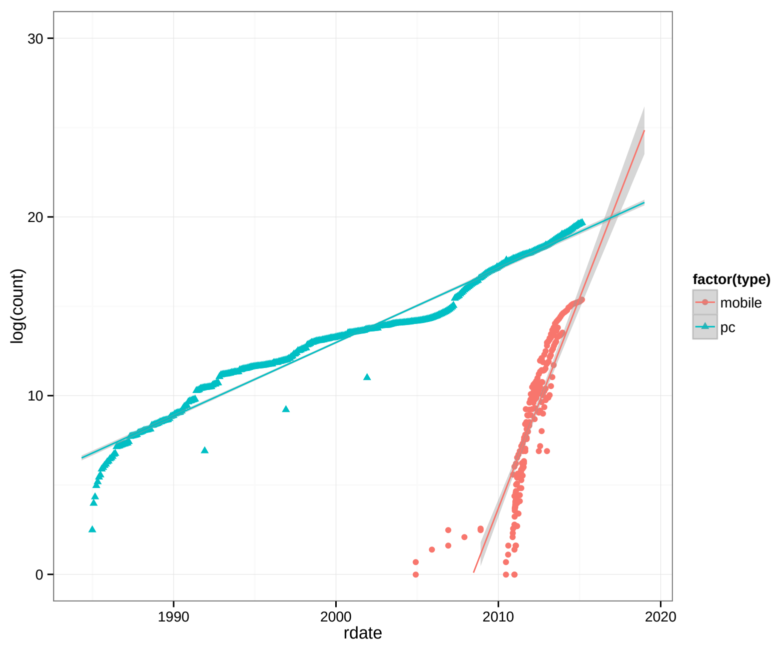


Figure 3.7: **Growth of unique malware specimens over time: PC vs mobile.** (log scale) While coming into existence much later, the volume of mobile malware is increasing at a rate that is approaching PC malware volume asymptotically. The marks indicated discrete industry data on quantity of unique malware specimens, the lines represent both an generalized linear model this data (iteratively reweighted least squares) and a projection, and the gray area surrounding the lines indicated error margin in the regression line. Based on current data, mobile malware will surpass PC malware circa 2017.

malware, however mobile malware is growing at a faster rate. Projections indicate that, based on quantity, mobile malware is likely to surpass PC malware in 2017.

Of paramount importance is the relatively quick pace of technological sophistication that can be observed in mobile malware. Instead of attempting to rank-order the relative sophistication of various, typically incomparable, malware categories, we can use first-use of a technique as a proxy. That is, instead of attempting to prove that encrypted malware is more sophisticated that non-encrypted malware, we simply say that encrypted malware appeared later and we therefore consider the technique more sophisticated than non-encrypted malware. Under this premise we can plot the first occurrence of various techniques used in malware which is shown in Figure 3.8. In PC malware we see early evolution of malware beginning slowly in the mid 1980's growing over time. Each category of malware shown has a complementary data point in the first occurrence of the category in mobile malware. Each category has been observed in mobile at a much quicker pace than in the PC—roughly three times as quickly.

The shortened timespan of technological advancement depicted in Figure 3.8, while notable, does not account for the sharp and continued acceleration of mobile specimens in Figure 3.7. One explanation of this discrepancy might be caused by certain dissemination models for mobile malware that are less common in PC malware. For instance, mobile malware is often formed by grafting malicious content onto legitimate applications (i.e., repackaging). Then

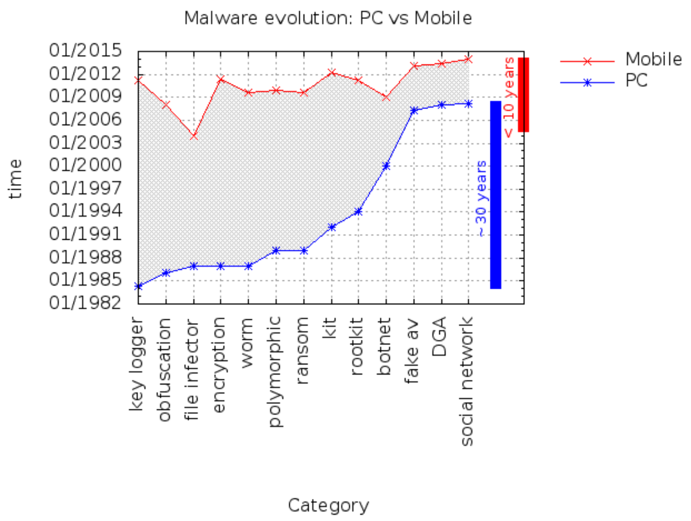


Figure 3.8: **Initial occurrence of malware by type: PC vs mobile**: Using time of occurrence as a proxy for relative sophistication, PC malware has evolved for more than 30 years; mobile malware is approaching the same sophistication in less than 10.

the newly crafted malware is distributed via malicious markets [225, 249], email or SMS phishing [58], or infected websites. In order to both entice users into installing the malware and to keep the malware installed on the mobile device, the original, legitimate functionality must remain functional and current. For this reason, the repackaging of the malware may keep pace with updates of the donor application. When operating a malicious marketplace, such updates might be regularly required for thousands of applications.

Inter-species: PC hybrid threats

In addition to the advancing pace of independent mobile malware sophistication, new PC-Mobile hybrid threats exist. These hybrid threats have appeared over the past several years to effectively form a complete graph of interaction. That is, not only are infected PCs used to infect mobile devices, but infected mobile devices are used to infect PCs. Malware on the PC and mobile device are designed to work in conjunction to achieve otherwise impossible tasks. Additionally, we see computers consciously employed to attack mobile devices and vice versa.

In September 2005, CardTrap became the first multipartite malware targeting mobile devices. The Symbian malware could implement installers for other Symbian malware (such as Cabir, Skulls or CommWarrior) but additional functionality enabled targeting Microsoft Windows computers. The malware overwrites many core Symbian applications and infects removable media with not only Windows Malware (such as Padobot or Kangen) but also `Autorun.inf` files. The `Autorun.inf` was intended to make the Windows malware automatically execute when the media is inserted into a running Windows computer. CardTrap was so invasive that some of the file infection would corrupt required files rendering the infected device unusable after it was rebooted.

Eight years later, in February 2013, DroidCleaner / Superclean malware also employed the `Autorun.inf` method to target Windows computers. This Android malware would infect computers by placing an `Autorun.inf` file on the SD Card, thereby automatically executing a backdoor on the PC when the SD Card was inserted into the PC or when the Android device was connected to a PC in media sharing mode. This malware comes long after Autorun features were disabled as a default, decreasing the utility of such attacks on updated systems. For example, Windows 7 has never enabled Autorun by default, and Windows XP disabled Autorun by default via an update in October 2012 [49].

However, the inter-species interactions are not limited to infected mobile devices attacking Windows-based PCs. Wirelurker malware discovered in November 2014 has the ability to infect iOS devices when they are connected to an infected OSX computer. In addition to information theft, Wirelurker automatically generates and installs malicious applications to an iOS device. Unlike most iOS malware, Wirelurker does not require the iOS device to be jailbroken as a pre-condition.

In some cases, established botnets have been used to disseminate mobile malware. Such was the case with Stels Android malware which was sent to would-be victims via the Cutwail botnet. Cutwail is a botnet for hire, comprised of infected Windows machines (perhaps millions) [210]. Most of the observed Cutwail campaigns typically involve sending SPAM e-mail, as was the case with Stels malware. Email purporting to be from IRS would entice a victim into visiting a malicious URL, which, when visited using an Android device, caused Stels malware to be installed.

Cutwail has also been known to disseminate banking malware, such as Zeus. In the past few years, every major botnet targeting the financial vertical has established a complementary mobile component. Zeus, Spyeye and Tatanga have "in-the-mobile" counterparts respectively known as Zitmo, Spitmo and Tangmo [248]. The most common use-case for these "in-the-mobile" malware is to circumvent two-factor authentication in concert with malicious transactions issued from an infected PC.

In addition to infected PCs, mobile malware is also disseminated via infected websites. When a miscreant infects a website, it may appear visually the same to a visitor but have hidden code that interacts with a browser to infect a victim computing device. This type of infection vector is now common for mobile malware dissemination. For example, one dissemination method for NotCompatible malware is via website of popular "blog" software such as Wordpress (NotCompatible is examined further in section 3.4.2).

3.4.2 Microevolution of Mobile Malware

In addition to evaluating evolution at a macro scale, various microevolutions can be studied. In this section we investigate microevolution by analyzing the evolution of two malware families: NotCompatible, and Svpeng. Such analysis can be performed on many malware families, we select NotCompatible simply because we have studied

Command	Operation
0	initial connect
1	initiate TCP proxy
3	shutdown existing proxy
4	send comms check
5	received comms check
253	set TCP proxy timeout
254	update C2
255	update backup C2

Table 3.3: **NotCompatible malware client commands.** These are part of a binary MuxPacket protocol, the first byte of the MuxPacket payload is the NotCompatible command, followed by data related to the command.

the family previously. Svpeng was selected for reporting because the exhibited evolution is dissimilar to that of NotCompatible.

NotCompatible family

First discovered in May 2012 [189], NotCompatible malware marked an early discovery of an Android botnet with a novel feature for mobile devices: the ability to proxy Transmission Control Protocol (TCP) connections. At the time, NotCompatible malware employed relatively simple C2 methods (see Table 3.3) to update communications configuration and to setup proxy connections. Unlike some other Android malware, NotCompatible is able to implement its features without requiring root privilege escalation. Upon receipt of a "1" command from a C2, NotCompatible creates a TCP proxy using Java's New Input/Output (NIO) class, which is readily available via standard Android APIs.

We have been able to monitor portions of a NotCompatible botnet using devices that we purposefully infected. At a high level, the infection process includes visiting a malicious website which in turn attempts to install the NotCompatible mobile application to the Android device. Investigation reveals that the malicious websites are themselves often victims of attack in that these websites are legitimate websites that have had a malicious iframe inserted at the very end of the HTML code. This iframe includes content from a secondary site. We experimentally determined that when an HTTP GET request is issued to this secondary site, if the HTTP USER-AGENT does not contain the (case-insensitive) string "android" the server will reply with an HTTP code 404. However, for browsers emitting a USER-AGENT containing "android," as the standard Android browser does, the secondary server will redirect to a third server. The tertiary server performs a similar server-side USER-AGENT check, and subsequently delivers the NotCompatible application packaged as a typical Android Application Package (APK) file to the device (example URLs in Figure 3.9).

We also later observed similar infection patterns except that the initial website is not a victim website, but instead is a lure. In this case, the website (URL sent via SPAM) will display a fake news site unless the USER-AGENT includes "android," in which case the browser is redirected via HTTP 302 to a secondary server. In some cases the secondary server will also perform a USER-AGENT check, then perform a Javascript redirect to a third server which

```
<iframe style="visibility: hidden; display: none; display: none;" src
    ="http://www.domain.tld/lll/lllllll"></iframe>
http://seconddomain.tld/fixup.php
http://seconddomain.tld/fixup2.php
```

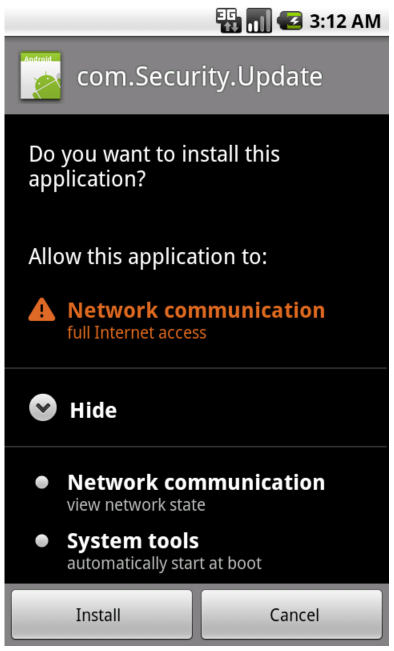
Figure 3.9: **NotCompatible infection flow**: Each line represents an established HTTP connection, in order. The initial interaction occur by visiting and infected website (depicted as domain.tld) to which a miscreant has added a hidden iframe. The victim client is then redirected (via HTTP 302) to a second domain (depicted as seconddomain.tld) and again redirected (via Javascript) to a second URL at the second domain. Despite the .php extension, this final URL (fixup2.php) is an Android APK—which is the NotCompatible malware. "tld" refers to any top level domain. Repeated lowercase "L" (l) represents any lowercase letter.

will deliver the NotCompatible APK. The two different methods of redirect might be employed in attempt to hinder automated sandbox analysis.

It is important to observe that the delivery of malware is not quite as automatic as is often found with PC malware. In PC malware, this would often be termed a "drive-by" installation, as the malware was automatically installed on the PC after visiting a malicious website. In the case of NotCompatible, the application is delivered as a standard web-based download. Therefore, two distinct user actions are required prior to a successful infection. First the user must allow applications to be installed from locations other than Google Play. The user may have actually enabled this setting long before visiting a malicious website and therefore this may be considered a precondition. Second, the user will be prompted to permit the installation of the malware (Figure 3.10). To increase the likelihood of the user completing the install, the miscreants give the application appealing file names such as `SecurityUpdateService.apk` which is delivered from domains such as `ANDROIDCLOUDSECURITYUPDATE.SU`. A weak form of social engineering that, based on observed infections, is patently successful. Once installed, the charade continues in that NotCompatible often runs as a Android service named `com.Security.Update` (Figure 3.11).

Once installed, the NotCompatible application takes initial action when the device is unlocked or upon reboot (the application has handlers registered for these broadcasted Intents). The malware then retrieves an embedded file which contains an Advanced Encryption Standard (AES) encrypted string containing a primary C2 domain, a backup C2 domain, and associated TCP ports. The NotCompatible application will then connect to a remote C2 (i.e., "phone home") reporting the type of connection to which the device is currently connected (e.g., WIFI or cellular). Once connected, the remote C2 may now issue commands to the infected device as enumerated in Table 3.3. That is, initiating a TCP proxy, performing a communications check, or replacing the embedded C2 information so that subsequent control is handled by a different server.

From a technical perspective, NotCompatible malware does not violate security properties established by Android. It does not attempt to access private data, break out of an application sandbox, gain permissions that are not defined in the application manifest or another other such violation. In most ways, it simply operates as any application would. In addition to successfully operating within the security boundaries of Android, NotCompatible did not originally


3:12 AM

com.Security.Update

Do you want to install this application?

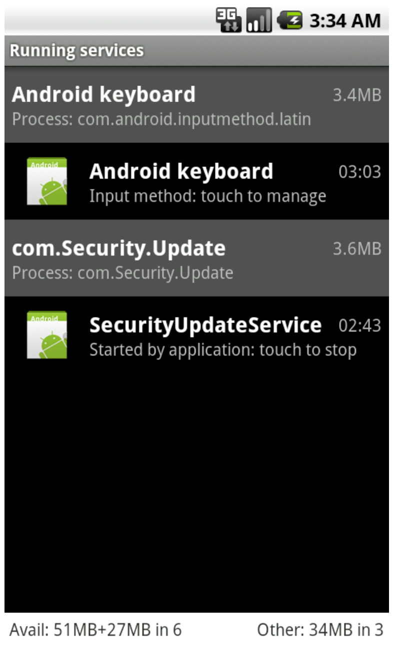
Allow this application to:

Network communication
full Internet access

Network communication
view network state

System tools
automatically start at boot

Figure 3.10: **NotCompatible install prompt**: Once the server-side concludes that the target is indeed an Android device to be victimized, the NotCompatible malware is delivered to the device. NotCompatible is packaged as a typical app, however since it is not distributed via Google Play, the "allow installs from unknown sources" setting must be enabled on the victim device. Even then, Android prompts the user to accept the install, as depicted here. Perhaps also of interest is the requested permissions, only three: INTERNET, ACCESS_NETWORK_STATE and RECEIVE_BOOT_COMPLETED. As of Android 5.0 (API 21), INTERNET is no longer presented to the user (implicitly permitted).


3:34 AM

Running services

Android keyboard 3.4MB
Process: com.android.inputmethod.latin

Android keyboard 03:03
Input method: touch to manage

com.Security.Update 3.6MB
Process: com.Security.Update

SecurityUpdateService 02:43
Started by application: touch to stop

Avail: 51MB+27MB in 6 Other: 34MB in 3

Figure 3.11: **NotCompatible infected device**: Once installed, NotCompatible runs as a standard Android service—posing as security software in effort to deceive the user into letting the malware persist on the device.

employ any code obfuscation techniques. This is only unusual because many applications at this time (including malware) employ obfuscation. Of course, obfuscation might be considered a desirable feature by malware authors wishing to hinder analysis, but it is also increasingly more common among typical applications due to default settings in the standard Android development environment.

In the next two years, the botnet continued to operate and the malware underwent iterations of increasing sophistication. In the months following initial discovery, relatively standard obfuscation was implemented and the network traffic, originally a plaintext protocol, gave way to encrypted data streams. Otherwise, at the device, the malware operates in much the same way as it did in 2012.

The most recent changes, observed in 2014, involve changes to the communications mechanisms which have evolved to a more resilient architecture. First, on the server-side, more checks are performed and a geographically distributed C2 approach has been employed. After the infected device initiates the initial connection to the embedded C2 the remote service will check the USER-AGENT and additionally check the source Internet Protocol (IP) of the connection against a blacklist. Assuming the C2 checks are satisfied, the C2 will now always deliver a new C2 configuration effectively acting as a redirect to other server to act as a real C2. These secondary C2s are assigned based on the physical location of the device. Furthermore, after the real C2 takes over from the redirecting C2, the infected device receives a list of other infected devices, effectively adding a Peer-to-Peer (P2P) component. If the real C2 to which an infected device is connected becomes unavailable, a new C2 may be gleaned from a peer infected device.

In the end, as far as we know, NotCompatible has never been used to directly steal information from an infected device, has never used its proxy capability to attempt to circumvent "internal" enterprise networks via WIFI, has never used rooting exploits, and has never used automated install techniques (such as browser exploits). Instead, the nefarious activity realized by the owner of an infected device is simply proxied network traffic—we observed primarily spam campaigns. Even so, the technological advancement, over two years, provides interesting data points in tracking relative sophistication of mobile malware.

svpeng family

Mobile malware called svpeng (a.k.a Crosate or cxeqem) is packaged as a typical Android application. Devices are infected when the application is installed by the user, often when prompted while visiting pornography websites or via SMS SPAM [220]. The application is typically presented as an Adobe Flash player in an effort to deceive victims about the true nature of the malware. We have also observed `com.android.pfx12` in addition to flash-based names like: `com.adobe.flashplayer`, `com.adobe.flashplayer`, or `com.adobe.flp`.

When a victim is actively using Google Play, the malware prompts a plausible modal dialog requesting the user to enter credit card details. Similarly, when a victim is actively using a banking application, the malware prompts a

```
private String AvEuck()
{
 return Settings.Secure.getString(getContentResolver(), "android_id");
}

String str2 = localTelephonyManager.getDeviceId();
if (str2.indexOf("000000000000000") != -1)
   System.exit(0);
if (str2 == null)
   str2 = AvEuck();
```

Figure 3.12: **Svpeng evasions**: Svpeng malware performs runtime checks for IDs specific to the Android emulator. If the malware detects that it is running in an emulator, it prematurely exits to thwart analysis. (note: the 'E' character in AvEuck is actually an 'F' and is obscured here for obvious reasons.)

plausible modal dialog requesting the user to enter the username and password associated with the bank account. In either case, the malware exfiltrates the stolen information to a remote server. The modal dialogues are implemented as a screen overlay, perhaps further suggesting to the victim that the dialog prompt originated from the application the user expected (Google Play or the banking app). Svpeng sends information remote servers using HTTP parameters.

Most versions of Svpeng employ some crude emulator detection in order to evade analysis. For instance, detecting the emulator-specific `deviceID` as seen in Figure 3.12. Most versions also attempt to gain the `DEVICE_ADMIN` capability by crafting an Android Intent claiming to be a Flash plugin installer.

Later versions employ a ransomware feature, claiming that authorities have locked the device. The malware then solicits payment via MoneyPak. These later versions also make use of Android "resources" to store images, signatures, and html files. The use of resources is not strictly a malicious advancement in sophistication, however, it does further demonstrate the general technological progression of the malware.

The svpeng malware author employed versioning in the malware. However, based on manual code comparison, it appears that the chronology of the versions we analyzed is not strictly ascending. The extracted version information in likely order of creation is as follows; 5.6, 5.91, 5.92, 5.9, 5.92, 5.93, 5.95, 5.97c, 5.7, 6.0cs, 6.00sc, 6.0s, 5.94, 8.0, 8.1, 8.2, 8.10, 8.21, 1.0, 1.1.100,1.1.200,1.2.100,1.2.200,1.2.300,1.3.300. In particular, note that 1.0 appears after 5.x, 6.x and 8.x versions. At this point, it is clear that the codebase for svpeng was notably altered.

Earlier versions of svpeng employ little obfuscation to hide the application's inner-workings. Though, an unused class "Cryptor" exists in the malware possibly indicating some intention employ encryption. These early versions also exclusively target Russian banks (specifically package name `ru.sberbankmobile`).

Recent versions of svpeng (1.0 or $1.x.y$ where $x \in \{1,2,3\}$ and $y \in \{100,200,300\}$), the malware employs slightly more sophisticated attempts in obfuscation such as the server base64 encoding data and statically defined substring replacement as shown in Figure 3.13.

Beginning with version 1.1.100, svpeng malware observes the presence of a number of financial applications

Package Name
com.usaa.mobile.android.usaa
com.citi.citimobile
com.americanexpress.android.acctsvcs.us
com.wf.wellsfargomobile
com.tablet.bofa
com.infonow.bofa
com.tdbank
com.chase.sig.android
com.bbt.androidapp.activity
com.regions.mobbanking

Table 3.4: **Svpeng target applications.** Package names of the applications that svpeng malware observes on an infected device. Presence of discovered applications is reported back to the remote attacker.

```
new Intent("android.app.action.EUCKERS".replace("EUCKERS",
    "ADD_DEVICE_ADMI=").replace("=", "N"))

localIntent.putExtra("android.app.extra.DEVICE_12345".replace("1",
    "A").replace("2", "D").replace("3", "M").replace("4", "I").replace("5", "N")

.indexOf("!eviceXdminAdd".replace("!", "D").replace("X", "A")
str = ".!Z2OceAdminAdd".replace("!", "D").replace("Z", "e").replace("2",
    "v").replace("O", "i");
```

Figure 3.13: **Svpeng obfuscation**: Each call to "replace" results in substring replacement of characters. In the end, `ADD_DEVICE_ADMI=` becomes `ADD_DEVICE_ADMIN`, `DEVICE_12345` becomes `DEVICE_ADMIN`, and `.!Z20ceAdminAdd` becomes `DeviceAdminAdd`. Obfuscation techniques like these limit `strings`-based analysis and may hinder static analysis systems that cannot reason about the effect of replacements.(note: the 'E' character in EUCKERS is actually an 'F' and is obscured here for obvious reasons.)

(denoted in Table 3.4) and reports results back to the C2 server. We have not been able to observe any change in behavior as a result of executing svpeng with any of the apps in Table 3.4 installed, therefore we simply theorize that the miscreants are profiling their victim pool for future financial fraud activities. Based on the apps of interest, it also appears that the miscreants are expanding area of operation from solely Russian to also include western financial institutions.

3.5 Related Work

The prevalence of mobile malware has been challenged, with some measuring very few instances. However, sharp measurement limitations accompany such studies, making the results of each a, sometimes severe, lower bound. For instance, in a study of a large US cellular provider conducted by Lever et al., the authors conclude that less than 1% of devices may be infected [141]. However, the measurement techniques in this study have no ability to observe traffic transiting other cellular networks, any non-cellular networks, any Virtual Private Network (VPN), or any proxy. Given that mobile malware may exhibit geographic localization, temporal transience, or the ability to alter behavior based on network connection type (see section 3.4.2), clearly many active infections will not be observed. The study is further limited to only malware samples that utilize DNS (during measurements).

Zhou et al. conducted alternative market research focusing on four alternative marketplaces [249], and have found a substantially smaller percentage of malware than we observed. Different from Zhou et al.'s study, we investigate a larger number of application marketplaces, and look at possible indicators of suspicious markets (e.g., extensive reuse of identical signing certificates). In another work, Zhou et al. focus on detecting repackaging in six alternative marketplaces [247]. The repackaging found in these six marketplaces was predominately performed in order to redirect ad revenue, but in a few cases the authors observed malicious payloads.

In [248], Zhou et al. describe a malware collection consisting entirely of Android samples. The authors provide measurement of the malware collection and describe the "evolution" of malware by studying related samples chronologically. The authors also find a large amount of application repackaging and provide measurements of activation mechanisms, secondary payloads, and permission used by malicious applications.

Burguera et al. also cite the repackaging and distribution in alternative markets as evidence for the need of their primary contribution in [69], which is a system for crowd-based behavioral malware detection. Both our work and that of Burguera et al. observe that applications are signed and the current signing process in no way inhibits repackaging and republication of applications.

Building upon the observation that repacking and republication of applications is not deterred by current signing processes, Lindorfer et al. conducted a large measurement study [144] of alternative markets similar to our measurements presented in section 3.3. More than 300,000 applications were downloaded facilitating longitudinal measurements of 20,000 unique applications over a three-month period. Across 16 alternative markets, the researchers were able to observe market proprietor behavior and inter-market relationships. For instance, more then 1,500 applications were removed from specific marketplaces during their study. Analysis of these removals revealed that nearly 8% of applications appeared in a second market only after being removed from another. Observing the removal patterns also gives insight into proprietor policing strategies and capabilities. Clearly, market proprietors behave differently. For instance, Google Play appears to detect and remove malware within days. Conversely, in the three months, Google Play removed 1,281 applications and all other measured markets combined only removed 384 (using Google Play as a reference, one might expect 19,215 removals across the 15 other markets). Other conclusions Lindorfer et al. lend independent confirmation of those drawn from our measurements. For example, their measurements indicate that some markets disseminate large proportions of malware.

Lindorfer et al. also claim that some markets "allow the publication of malicious applications." However, it is not clear if this claim is a permission via policy or simply a lack of policing on the part of the market proprietor. Similar to our measurements, those by Lindorfer et al. reflect a strict lower bound on malware identification. Unlike our measurements, potentially unwanted applications (or "greyware") were included in their reporting. Regardless, their measurements demonstrate that malware is downloaded an order of magnitude more that benign applications and that some of the 16 markets exhibit strikingly similar application corpora to others. For instance, the andapponline market shares 47% of *identical* applications with the opera market.

3.6 Discussion

Application markets are commonplace for mobile devices. In addition to establishing marketplace measurements, demonstrating the rather large problem of mobile malware, we have shown that not all markets are created equal: quite the opposite, in fact, as some distribute malware almost exclusively. Conversely, Google Play exhibits a remarkably low rate of mobile malware, though our measurement does only reflect a strict lower bound.

By analyzing signing certificate strategies from our market measurement data, we observed that markets that deliver the highest percentage of malware are also those that reuse signing keys the most, unilaterally across the marketplace in fact.

In addition to the rapid technological advancement of smartphones compared to the PC, we can measure advancement of mobile malware compared to PC malware. For instance, mobile malware is quantitatively growing at a more rapid pace than PC malware, and is poised to overcome PC malware in the near future. From a feature perspective, mobile malware is also rapidly re-implementing techniques previously observed over decades of combating PC malware.

Evolution can also be observed at a much more granular level of abstraction, as is the case with the two malware "families" we detailed in this chapter. In these cases, clear technological advancement of employing encryption, obfuscation, novel C2 techniques, and increased range of targets can clearly be observed in progressive instances of malware.

While PC and mobile malware are often considered to be disjoint sets, victims often engage in both domains. For this reason, miscreants have forged new types of hybrid attacks where victim PCs may be used to target smartphones, or vice versa. Indeed, miscreants make coordinate attacks through infected PCs and smartphones working in concert.

A popular aspect of biological evolution not yet mentioned in this chapter is the concept of safety provided by diversity. Research regularly demonstrates that diverse populations are more resistant to infection. Applied directly to the mobile ecosystem, one might look to the two most prevailing platforms: Android and iOS. Android is regularly shown to have a few fundamental problems, among them are lagging updates and fragmentation (as depicted in Figure 2.3. Fragmentation, necessarily implies diversity. Further, it is possible to be "up-to-date" in different software branches of Android. For instance, a Honeycomb (3.2.5, Jan 2012) device might be fully updated from a security perspective even though Ice Cream Sandwich (4.0, Oct 2011 [157]) has already been released.[8] Conversely, iOS is

[8]Android update methodology with respect to branches is discussed in the documentation found at `https://source.android.com/source/code-lines.html`

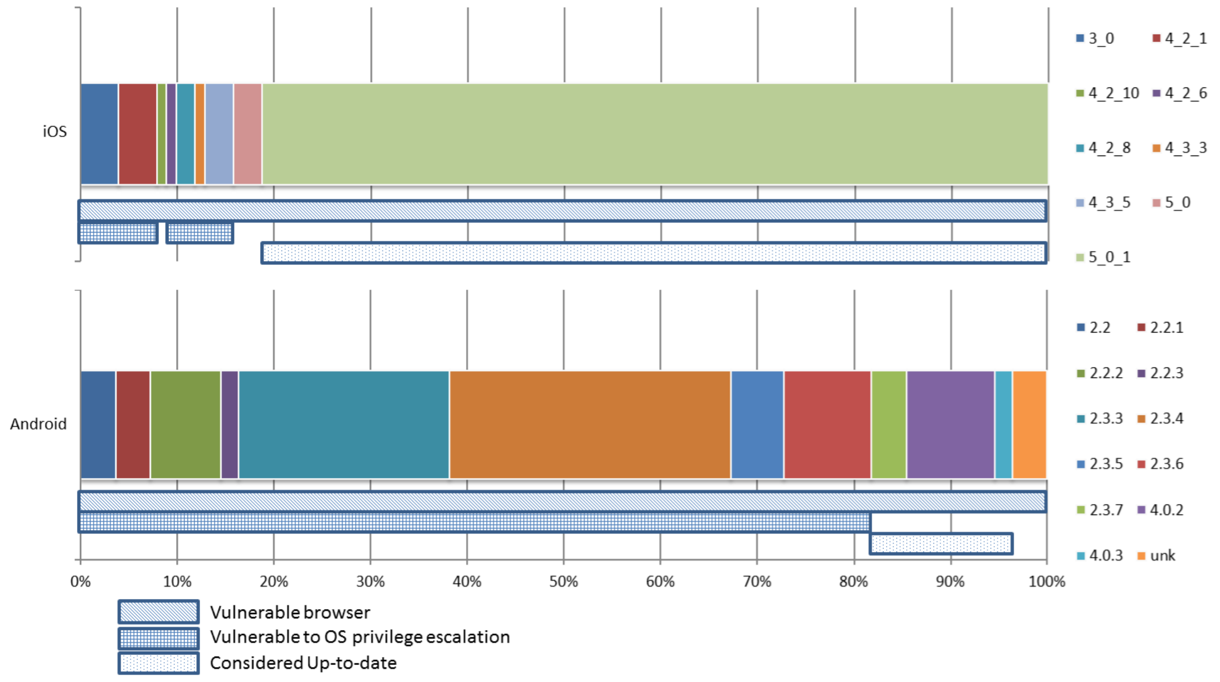


Figure 3.14: **Android and iOS version proportion**: Android exhibits higher proportions of many OS versions, whereas iOS exhibits a single, clear majority. At the time of the study, all participants were likely using a vulnerable browser. Further more than 80% of Android and 15% of iOS participants were likely susceptible to a root privilege escalation. In both iOS and Android, more than one OS version may be considered up-to-date. Three Android versions were considered up-to-date at the time of the study.

oft-cited as a less diverse system. Figure 3.14 depicts the proportion of Android and iOS participants encountered in a security study [228]. Android clearly exhibits a more diverse system, no single Android version has a majority, with most only comprising of a few percent. However, nearly all of the mobile malware currently targets Android. In this way, mobile malware does not appear to follow the common aspect of safety through diversity.

Chapter 4

Developer Protections

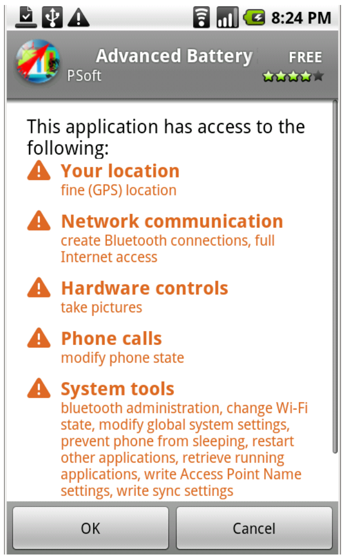
Smartphone users are regularly required to make security-conscious decisions when installing and using mobile applications. These decisions are often only informed with very limited data reflecting what the developer has *specified*, not what the application actually *requires*. In this chapter, we focus on preventing well-intentioned developers from creating over-privileged software.[1]

Developers for the Google-backed central repository for mobile applications, Google Play, enjoy more freedom than on the Apple-moderated App Store for iOS applications. A would-be Android developer need only register for an account and pay the $25 fee. Further, Android applications can be installed from third party websites circumventing the Google Play altogether. Users historically make poor privacy and security decisions especially when a warning is difficult to understand and/or acts as a barrier to immediate gratification [51]. Like many Google products, Android seeks to limit the volume of privacy and security-related warnings presented to the user. Application privacy and security settings are accepted by the user prior to install, and, prior to Android 6 (API 23), the user typically never again has to make a privacy or security-related decision pertaining to that application.[2] An example of an install time prompt can be seen in Figure 4.1 in which a battery monitoring application is requiring access to the GPS device, phone state, Bluetooth, Internet access and a multitude of other permissions. The install time permission requirements are determined by the developer and may or may not accurately reflect permissions that the application actually requires for proper operation.

While the Android platform already provides copious amounts of information to developers as well as a reasonably rich development environment, there is no straightforward way for a developer to determine appropriate permissions to request. Developer specified permission requirements may in fact be a superset of the permissions the application actually requires resulting in violation of the principle of least privilege [198]. Least privilege is an important aspect of system design, benefiting system security and fault tolerance. The main contributions of this chapter are to de-

[1] Portions of this chapter previously appeared in [227].

[2] Unless the permissions change between version upgrades, in which case the user will be prompted once for the new version's permissions.


8:24 PM

Advanced Battery FREE
PSoft

This application has access to the following:

Your location
fine (GPS) location

Network communication
create Bluetooth connections, full Internet access

Hardware controls
take pictures

Phone calls
modify phone state

System tools
bluetooth administration, change Wi-Fi state, modify global system settings, prevent phone from sleeping, restart other applications, retrieve running applications, write Access Point Name settings, write sync settings

Figure 4.1: **Android permission prompt during application install** Why does a battery application require so many invasive permissions? Figure from [227].

scribe a tool, Permission Check Tool, that aids the developer in specifying a minimum set of permissions required for a given mobile application, and to describe the creation of an associated API-permission database for Android. The tool analyzes application source code and automatically infers the minimal set of permissions required to run the application. This approach is unique in the method used to determine permission requirements and, to encourage adoption, is implemented alongside the existing Integrated Development Environment (IDE) recommended for Android development.

First we provide background on Android's permission model in section 4.1, expanding upon section 2.2.1. In section 4.2 we describe the inner workings of the Permission Check Tool. We present related work in section 4.4, provide some avenues for future work in section 4.5, and have a concluding discussion in section 4.6.

4.1 Background

Built upon a Linux kernel, Android uses OS primitives (such as processes and UIDs) as well as a Java Virtual Machine (Dalvik) to isolate applications providing a safety sandbox [205]. The Android platform introduces about 130 application level permissions (see Table 2.1) that are requested at install time and silently enforced any time the application is executed [75, 171, 203]. Unlike other privacy models, such as that employed in iOS, the user is not prompted when an application requests a resource during execution.[3]

These application level permissions generalize access controls into categories like "access location information" or "access the network." In some cases this generality may force an application to request more access than needed.

[3]Prior to Android 6 (API 23) as discussed in section 2.2.1.

Consider, for instance, an Internet radio application that only needs access to a single URL on a single domain, but must request access to all network resources. Other applications request large sets of controls, many of which intuitively have little or no use to the advertised functionality of an application. The ambiguity, generality and misuse of Android application permissions have led to statistics such as "50% of applications send info to third parties" [191]. This information disclosure is asserted to be "clearly wrong" [191] even though the practice is likely in accordance with Google's privacy policy [38].[4] Other studies claim 1 in 5 applications are a privacy threat, 1 in 20 can make arbitrary phone calls, 3% can arbitrarily send text messages, and 383 can access authentication information from other applications and service [224].

Users have become habituated to clicking through terms of service and warnings, and Android users are thus unlikely to pay much attention to notices about Android permissions. Average users cannot be expected to understand the semantics of approximately 130 permissions. Similarly, users likely tend to be unaware of the privacy impacts of their decisions. The disconnect between the user accepting security and privacy settings, once, during install and the enforcement of these settings upon subsequent application execution, along with ambiguous permission descriptions can lead to a fundamentally unaware user base. Given that there is no central moderator of the Android market, the policing of applications is largely left to these same users.

We suggest that applications that request more permission than needed fit into two general sets: applications that "do something 'questionable' or 'malicious' on purpose" (e.g., surreptitiously collecting information for advertising purposes), or applications that inadvertently request additional permissions due to lack of understanding, laziness, or expected future use of a capability. In the first case, the mobile application actually requires permissions that may seem unnecessary to a user. We take a simplistic definition of "proper operation" assuming that all API calls that require a permission are required for proper operation. For example, an application marketed as a battery monitor possibly *should* not require the GPS permission that the application requested. For our purposes, if the application makes use of the API interface for location, then the permission request is legitimate. However in the second case, when superfluous permissions are requested, the application is clearly requesting more permission that required, a plain violation of the principle of least privilege [198]. Note that under our definition, if the previously mentioned battery monitor application actually makes use of all associated API calls, it may actually have correctly specified permissions. Of course, such an application may not be classified similarly under other metrics such as End User License Agreement (EULA) or user expectation. If the second set was reduced to near empty, then users would only have be wary of "questionable" applications.

Much of the application-specific burden of identifying requested permission falls with application developers, who

[4] Google's Privacy Policy is fairly general and vague allowing for many forms of information collection and sharing. For example, the policy contains verbiage such as *"We may share with third parties certain pieces of aggregate, non-personal information," "we provide such information to our subsidiaries, affiliated companies or other trusted businesses or persons for the purpose of processing personal information on our behalf," "We may process personal information to provide our own services. In some cases, we may process personal information on behalf of and according to the instructions of a third party, such as our advertising partners,"* that allow for a wide variety information collection and sharing.

are required to specify permissions that an application will request. The Android platform has no straightforward way for a developer to determine appropriate permissions to request. Either the developer needs to observe in the online API documentation that a permission is needed by a particular function call, or, more likely, observe in the emulator that a Java error is thrown when such a function is called. This problem posed to the developer is compounded by the granularity and ambiguity of the permission mnemonics and associated descriptions. The permission `READ_SMS` is a fairly straight-forward and specific, but `INTERNET`, while straight-forward, is very general. Other permissions may be considered obtuse (`ACCESS_SURFACE_FLINGER`)[5] or ambiguous (`DIAGNOSTIC`). Even the clear permissions have created serious user confusion, as evidenced by the user outrage when Rovio added the SMS permission to their popular game Angry Birds. The permission was required because Rovio had added mobile payment capabilities allowing in-game purchasing over SMS [167].

Even though applications obtained from the market are in a compiled form that does not readily permit source code analysis, the requested permissions can be extracted from the binary XML with relative ease. From a corpus of 34,000 free Android applications obtained from Google Play in March 2011, we observed indicators that developers are currently not specifying permissions according to least privilege. For example, more than four percent of the applications specify *duplicate* permissions. That is, the application manifest contained the same permission more than once (for some, many times). Table 4.1 shows the number of applications that request duplicate permissions by market category, demonstrating that even reference libraries and medical applications contain some duplicates. Clearly some categories are more afflicted with duplicate permissions than others, Photography, Racing, Sports Games and Cards & Casino Games, for instance. Table 4.2 shows the permissions most often duplicated.

4.2 Permission Check Tool

We created a tool that aids the developer in assessing appropriate application permission requests. To encourage adoption, this tool is implemented as an Eclipse IDE plugin that can be used alongside the Android-specific development environment provided in the Android Software Development Kit (SDK). The tool evaluates an Android application for platform permission access and informs the developer on minimum controls required for proper execution. In this section we discuss the creation of a permission-API database and the Eclipse implementation.

4.2.1 Permission-API database

We created one-to-many permission-API mappings by manually parsing the API documentation[6] and creating a database of functions and permissions upon which they depend. Some permission mappings are more complex than others. For example, instantiating and using a `BluetoothSocket` requiring the `BLUETOOTH` permission, is a

[5]The extended description of this permission is “Allows an application to use SurfaceFlinger’s Low Level Features.”
[6]While no longer automatically distributed, it can be locally installed using the SDK Manager tool.

Market category	Total apps	With duplicates	Percent with duplicates
Arcade and Action	1344	17	1.26
Books and Reference	1452	24	1.65
Brain and Puzzle	1352	14	1.04
Business	1092	38	3.48
Cards and Casino	842	85	10.10
Casual	966	4	0.41
Comics	838	11	1.31
Communication	1311	77	5.87
Education	1305	21	1.61
Entertainment	1522	40	2.63
Finance	1354	44	3.25
Health and Fitness	1258	36	2.86
Libraries and Demo	1156	21	1.82
Lifestyle	1489	48	3.22
Live Wallpaper	537	14	2.61
Media and Video	1360	49	3.60
Medical	527	2	0.38
Music and Audio	1124	89	7.92
News and Magazine	1419	63	4.44
Personalization	1342	54	4.02
Photography	1165	283	24.29
Productivity	1319	54	4.09
Racing	216	76	35.19
Shopping	1155	46	3.98
Social	1296	41	3.16
Sports	1433	23	1.61
Sports Games	365	71	19.45
Tools	690	27	3.91
Transportation	454	8	1.76
Travel and Local	1473	35	2.38
Weather	342	4	1.17
Widgets	1395	64	2.59
Total	34893	1483	4.25

Table 4.1: **Applications with duplicate permissions by market category.** Applications in every category exhibit duplicate permissions, even reference libraries and medical applications. Photography, Racing, Sports Games and Cards & Casino Games are relatively egregious offenders. Figure expanded from [227].

Permission	Count
INTERNET	620
ACCESS_NETWORK_STATE	438
READ_PHONE_STATE	153
RECEIVE_BOOT_COMPLETED	147
WRITE_EXTERNAL_STORAGE	59
READ_CONTACTS	49
ACCESS_FINE_LOCATION	48

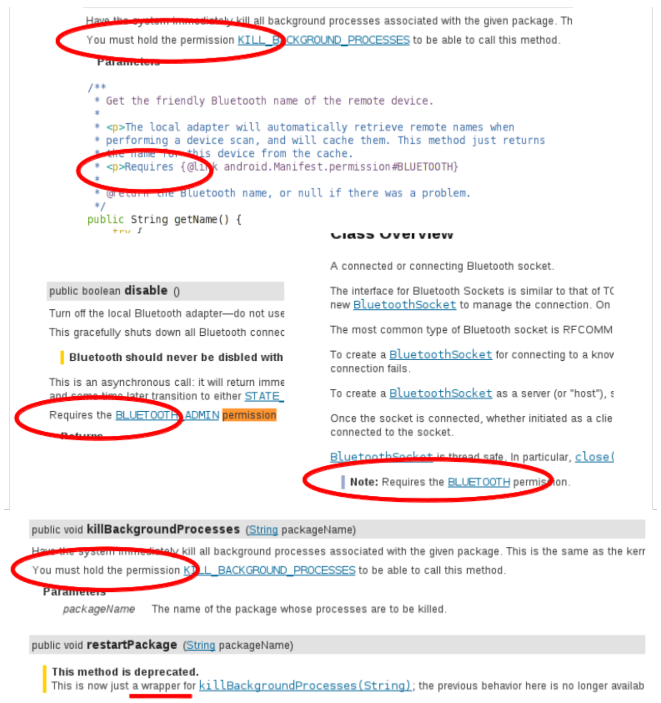
Table 4.2: **Top duplicate permissions requested.** `INTERNET` is clearly the duplicated, though those possibly related to identity (`READ_PHONE_STATE`) and privacy (`READ_CONTACTS`) are also present. Figure from [227].

fairly straightforward example, but the `LocationManager` class cannot be instantiated directly and the permission varies based on constants used in the instantiation: when using `GPS_PROVIDER` with `LocationManager` the required permission is `ACCESS_FINE_LOCATION`, when using `NETWORK_PROVIDER` with `LocationManager` the permission is `ACCESS_COARSE_LOCATION`.

Permission-API databases can, and should, be created a priori and simply loaded by the tool the first time the plugin is executed. Once created, the databases should require little maintenance since each database is particular to an Android API revision which is static. The only maintenance would be the result of an error or omission in the database itself.

Inconsistent nomenclature in the documentation further complicates the creation of a Permission-API database. Figure 4.2 shows several examples taken directly from the documentation. The class overview for `BluetoothSocket` shows the requirement for the `BLUETOOTH` permission in a "Note": "Requires the `BLUETOOTH` permission." Similarly the documentation for the `disable[]` function we see "Requires the `BLUETOOTH` permission" but not as a "Note." Other instances demonstrate more variation as is the case with `KILL_BACKGROUND_PROCESSES` which is called out with "You must hold the permission...to be able to call this method" in lieu of the more common "required." In Figure 4.2 we see that a permission required for `restartPackage` is actually noted in the text associated with `killBackgroundProcess`.

As a result of these complexities, the version of the database developed as a proof of concept for our tool handles all method (function) and class level permission enforcement denoted in the SDK documentation for Android 2.2



Have the system immediately kill all background processes associated with the given package. Th

You must hold the permission KILL_BACKGROUND_PROCESSES to be able to call this method.

Parameters

```
/**
 * Get the friendly Bluetooth name of the remote device.
 *
 * <p>The local adapter will automatically retrieve remote names when
 * performing a device scan, and will cache them. This method just returns
 * the name for this device from the cache.
 * <p>Requires {@link android.Manifest.permission#BLUETOOTH}
 *
 * @return the Bluetooth name, or null if there was a problem.
 */
public String getName() {
    try {
```

public boolean disable ()

Turn off the local Bluetooth adapter—do not use

This gracefully shuts down all Bluetooth connec

Bluetooth should never be disbled with

This is an asynchronous call: it will return imme and some time later transition to either STATE_

Requires the BLUETOOTH_ADMIN permission

Returns

Class Overview

A connected or connecting Bluetooth socket.

The interface for Bluetooth Sockets is similar to that of TC new BluetoothSocket to manage the connection. On

The most common type of Bluetooth socket is RFCOMM

To create a BluetoothSocket for connecting to a know connection fails.

To create a BluetoothSocket as a server (or "host"), s

Once the socket is connected, whether initiated as a clie connected to the socket.

BluetoothSocket is thread safe. In particular, close(

Note: Requires the BLUETOOTH permission.

public void **killBackgroundProcesses** (String packageName)

Have the system immediately kill all background processes associated with the given package. This is the same as the kerr

You must hold the permission KILL_BACKGROUND_PROCESSES to be able to call this method.

Parameters

packageName The name of the package whose processes are to be killed.

public void **restartPackage** (String packageName)

This method is deprecated.
This is now just a wrapper for killBackgroundProcesses(String); the previous behavior here is no longer availab

Figure 4.2: **Android documentation permissions examples** The circled areas call out various mechanisms for calling out required permissions. Figure from [227].

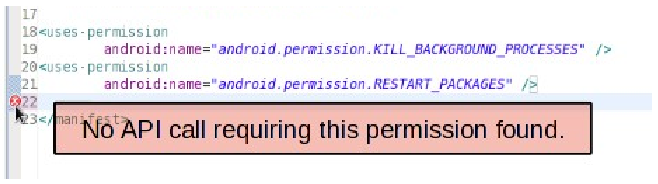
(API 8). The database consists of two sets of one-to-many mappings: permission associated with one or more methods and permission associated with one or more classes. Permissions that have no associated method (or class) have no entry in the set.

4.2.2 Static Analysis

When the user invokes the plugin, the plugin parses application requested permissions from `AndroidManifest.xml` into a map. Each map entry stores meta information about permission, such as source code line number, and is initially marked as *unused*. The plugin then inspects all Java source files (Eclipse `IProject "members"`) using Eclipse's built in API functions for Java, in particular the Abstract Syntax Tree (AST). Each API reference is checked against the Permission-API databases, if a permission is found to be required for a reference, the associated map entry is marked as *used*. Once all source has been inspected, permission entries in the map still marked as *unused* are known to be extraneous. As an additional aid to the developer, references that are found to require a permission that was not specified in the `AndroidManifest.xml` are also tracked in order to suggest additional permissions for inclusion in the manifest and prevent error conditions during execution.

4.2.3 User Interface Notification

After the static analysis completes, both permissions that have been specified by the developer but are not required and permissions that are required by the application but have not been specified are known. The plugin again utilizes familiar Eclipse features to notify the developer of any omitted and/or extraneous items. An extraneous permission is shown in Figure 4.3 with a red "error" mark and associated tooltip text. The "error" condition is recognized by Eclipse and will prevent the build (unless the developer corrects the condition and reruns the tool, or manually clears the mark). Similarly, the tool will notify the developer, using a yellow "warning" mark, in the case where a function requiring a permission is used yet the developer has not specified the appropriate permission in the manifest. By using the existing Eclipse interfaces, other standard views, such as the Problems View, can be used to obtain a list of permissions-related errors for the entire Eclipse Workspace.



```
<uses-permission
        android:name="android.permission.KILL_BACKGROUND_PROCESSES" />
<uses-permission
        android:name="android.permission.RESTART_PACKAGES" />

</manifest>
```

No API call requiring this permission found.

Figure 4.3: **Eclipse plugin highlighting extraneous permission** The red tooltip is automatically placed by the plugin, bring the extraneous permission to the developer's attention. Figure from [227].

Application	Extra Permissions
droidcon	1
meshapp	2
posit mobile	7
selenium	1
wifi-tether	1
YouTube Direct	3

Table 4.3: Extraneous permissions found in open Android application source

4.2.4 Tool Results

Of the applications obtained directly from Google Play, only a very small fraction of the developers have elected to make source code available to the public, impeding empirical analysis of our source code oriented tool. In order to evaluate our tool beyond artificial exemplar tests, we searched the Internet for open source Android applications. We used the Google search engine to search, independently, for "android open source" and "androidmanifest.xml." From the search results, we gathered 45 applications, biased by Google's PageRank popularity, the search terms themselves, and our ability to manually interpret search results and locate actual Android application source. The dataset we created for evaluating our tool should not be considered rigorous or generally representative of Android applications found in online marketplaces.

Many mobile applications are developed by novice individuals or small groups that have minimal quality assurance or code auditing procedures. Even so, one might predict that, due to the open audit ability, applications that have available sources are more likely to have minimal permissions specified than the closed source applications. However, our dataset revealed that some of these open sourced applications indeed have or have had extraneous permissions specified. As shown in Table 4.3, our tool readily identified six (13%) applications to have extraneous permissions via source analysis. Not only did applications include superfluous permissions, but some specified the same permission multiple times [41,61] or specified fictitious permissions [39].

In observing permissions requested by applications (i.e. actually using Android devices and Google Play to install and use applications) we hypothesize that applications are likely requesting more permissions than required for proper operation. By readily locating several open source applications that indeed specify superfluous applications, confirmation of this hypothesis is at least plausible (though not confirmed in a representative manner to any particular online market). In an effort to confirm correct operation of our tool, we randomly selected ten applications from our corpus for manual analysis. Our manual analysis was to ensure that: (1) each permission listed in the `AndroidManifest.xml` was a genuine Android permission, (2) at least one associated API existed in the application source for each permission specified in the `AndroidManifest.xml`, and (3) the application didn't use any APIs that warranted inclusion of additional permissions. In order to evaluate use of APIs that may require an additional permission, we primarily resorted to actually using the application in an Android emulator instance and observing errors via the graphical interface or via emulator log messages. However, we also informally audited the source code.

Of the ten selected for manual analysis, our tool had only found superfluous permissions in one, Posit Mobile. Posit Mobile not only included permissions (`UPDATE_DEVICE_STATS` and `CONTROL_LOCATION_UPDATES`), specified in SDK documentation as "Not for use by third-party applications" (see section 2.2.1) but also several that are invalid for the API (`ACCESS_LOCATION`, `ACCESS_GPS`, `ACCESS_ASSISTED_GPS`, and `ACCESS_CELL_ID`). Through manual analysis of the application, we concluded that the `READ_PHONE_STATE` permission was required due to use of the `TelephonyManager` class, `VIBRATE` was required due to the presence of the `Notification` class, `SEND_SMS` was required due to Android's `SmsManager` method `sendTextMessage(),` `WAKE_LOCK` was required due to Android's `PowerManager` method `newWakeLock()`, `ACCESS_WIFI_STATE` and `CHANGE_WIFI_STATE` were required due to use of Android's `WifiManager` methods `getConnectionInfo()` and `setWifiEnabled().` In all, of the 13 permissions specified, six were legitimately required by the application, two were restricted use, and five were invalid for the API (`ACCESS_GPS` appeared twice in the manifest).

Posit Mobile is an application that uses hardware features to communicate with other devices and/or requires a dedicated server component. Due to these unusual constraints, we did not manually exercise functions of the application in attempt to ensure that at the permissions specified in the `AndroidManifest.xml` were at least those required for proper function. Instead we manually inspected source code and concluded that there were likely no missing permissions.

We inspected nine other applications in this way: HockeyAndroid [14], Soar Android [27], FootPath [11], Imperial Calculator [15], DuckDuckGo [8], Google Authenticator [13], Track Map [28], Yaxim Android [33], and Picasso [23]. All permissions specified in these applications were genuine permissions (i.e. all syntactically correct according to the SDK documentation). Each of these applications specified six or fewer permissions and every permission had at least one related API in the application source. So, none of these nine specified superfluous permissions. We exercised application functions in an Android emulator by manually interacting with the applications, attempting to exercise all readily available features (like Posit, such interaction was limited on HockeyAndroid, FootPath, Track Map, and Picasso). In this manual interaction we observed no crashes, emulator log errors related to permissions, or Graphical User Interface (GUI) prompts indicating that a potential permission error was caught via error-handling software. As with Posit Mobile, we furthered our analysis via informal source code audit, which revealed no APIs use warranting inclusion of any permission not already specified. Therefore, we identified no under-permissioned applications. This result is not surprising due to the low likelihood of sampling a functionally broken application in our set.

4.3 Limitations

An implementation of this tool was created as a proof of concept. Accordingly, the database of permissions and APIs were only created for a few versions of the Android API (e.g. 8 and 9). Further, we created the database manually, through source code and documentation audit. This process is time-intensive and error-prone. Any errors in the database may result in errors in the operation of the tool (i.e. an API omitted in the database may result in a permission being flagged as superfluous by the tool, even if the permission is actually required for proper function of the application).

Developers may elect to include binary components in applications. Inclusion of such code is not uncommon, and may introduce a variety of undesirable qualities into an application [177]. Further, inclusion of any binary component precludes source code analysis of that component, such as the analysis performed by our tool.

Our implementation was also created to specifically interoperate with the Eclipse IDE, so the implementation provided does not directly assist developers that build Android applications via other means.

Determining comprehensive error rates for our implementation not only requires a large corpus of source code, but also a systematic method of evaluating the "proper function" of an application. Current methods of exercising application operations are not comprehensive, therefore there is no reliable, scalable method of determining false-positive or false-negative errors.

4.4 Related Work

Several groups have explored Android security and attempted to formalize the security model [75,205], apply security enhancements found on modern computers to Android [170, 196, 202, 203], and adapt or extend Android's current models [97, 164, 171, 194]. Some of the general reviews of Android security also take into account characteristics specific to the mobile market, such as battery life or billing based on throughput [194, 203]. Related research spans across the underlying Linux base, the Android middleware and Android applications. Android applications have been studied in the context of a permission model and as case studies. Case studies often involve the collection and various analysis of a percentage of Google Play. This percentage varies widely in collection manner and number of applications used in the study (from 30 [97] to 48,000 [224]).

Twenty-five Firefox browser extensions were analyzed by Barth et al. [62] who found that 78% request more privilege than required, demonstrating a lack of least privilege in the browser extensions. Enck et al. scrutinize 30 applications demonstrating their dynamic taint system on Android. Of the 30 popular applications, 1/2 to 2/3 were found to exhibit questionable behavior such as reporting location to 3rd parties or disclosing sensitive information [97]. In this case the applications were clearly utilizing the permissions requested of the application.[7] Complementing this

[7]Enck et al. also note that the EULA and privacy policies, if provided, omit or are not clear about advertising and 3rd party data collection

work, we have focused on applications that request extraneous permissions neatly addressing both sets of applications mentioned in section 4.1.

We manually created our database from API documentation. Creating databases in this way is arduous. Work by Felt et al. describe a project that facilitates automating the creation of permission-to-API mappings [108]. Creating the databases in this way should be more robust and maintainable.

Perhaps the most related work is [201] which explores iPhone privacy issues in rooted iPhones, iPhones vulnerable due to software vulnerabilities, and fully patched iPhones that ostensibly only allow data access through the published API. The author releases an open source, proof of concept application: SpyPhone, which collects users data. Some of this collected data would likely be expected by a typical smartphone user: address book, phone call logs, email messages. Other bits of information may cause alarm in many users such as the keyboard cache file.

4.5 Future Work

Several updates and additions could be made to the tool. For example, the tool already integrates with the suggested IDE for Android development, but the process of creating an appropriate permission set could be automated as part of the build process. The permission-function pairs used by the tool need to be generated for all API levels. Additionally, there are several indirect enforcements of permissions through bit-field class data members, or class interfaces that are more difficult to extract from documentation. It may be difficult to enumerate all instances from documentation and API fuzzing may be more appropriate. Eclipse integration could also be improved through the use of an Eclipse *Nature* allowing for dynamic checking of extra and omitted permissions as software is developed. A Nature can also automatically detect when the user had corrected an error mark eliminating the need for the developer to re-run the tool. Using the Eclipse AST to locate API references is a convenient method of source inspection, but more advanced static analysis techniques could be employed.

Corpora of applications and application source code should be created from open sources and then analyzed using future versions of this tool. Such corpora will be useful for many future Android based research projects, both related to this work and not. In this work we assumed a very broad definition of “proper” in regard to the operation of an application. Applications such as a battery monitor need very few permissions, and certainly should not require Internet access, contact list access or location information in order to report battery information to the user. Further study into the privacy and security impacts of publicly available Android applications that offer similar “extended functionality” is warranted.

4.6 Discussion

Market share alone demonstrates the viability of the Android platform. Both the generality and granularity of application-level permissions warrant some concern to users and developers alike. However, even if the Android security and privacy models remain unchanged, developers can create applications that request only the minimum set of permissions required for execution. We have presented a tool that aids developers in just this way by highlighting code that require certain permissions (and thus avoiding errors during execution) and highlighting requested permissions that are not needed (maintaining least privilege, a desirable security feature). Using the tool requires no changes to the development process, application publication process, or the Android framework. Our tool assists developers in creating safer applications, both by lessening the access that is granted to an application and by reducing the number of permissions presented to a user.

The tool described here is intended as a source analysis tool for developer use. There is no requirement for developers to openly release the source code of Android applications. Compiled and packaged applications are typically a collection of binary XML and Java classes in APK format. As such, there is no ready dataset of Android application source code which makes extensive empirical analysis of this plugin difficult. Even so, we were able to easily locate six offending open source applications, comprising 13% of our sample.

The tool is implemented as a simple proof-of-concept demonstrating the viability of the approach. However, using the tool for production applications should present minimal risk. The plugin is available as a jar file[8] and can be installed by saving the file into the "plugins" directory for Eclipse.

[8]Source code available at `https://github.com/tvidas/perm-tool`.

Chapter 5

Market Protections

Software market proprietors seek methods to entice users to utilize their market over competitors. The metrics for such differentiation are endless, but it is easy to presume that factors such as application cost, selection, and variety play a role as well as market ease-of-use, social factors, and perhaps security. It is not a stretch to draw analogies to physical marketplace relationships and the notion of consumer trust. In this chapter, we turn our attention to aiding market proprietors in policing their market offerings.[1]

The number of applications available for mobile phones and tablets has surged dramatically over the past couple of years; this trend has been particularly pronounced for Android devices, which now represent three out of four mobile devices [116, 238]. Concomitant with this rise in the number of applications available, malware targeting mobile platforms, and specifically Android, has also started to appear [109, 225, 248]. The devious nature of malware impedes malware measurements, however, there is little doubt that the overall volume of mobile malware is increasing at a pace that makes it difficult to sustain systematic manual analysis (see section 3.4.1). It is, therefore, important to develop automated analysis capabilities for mobile malware.

Detecting, analyzing and combating mobile malware presents a number of unique challenges. First, different from the situation with personal computers, users generally do not have full administrative access to their mobile device, which makes it much more challenging to develop effective AV tools. Second, carriers and network operators, who can fairly tightly control the network, may have only limited capabilities to control individual devices. Third, techniques useful to “sandbox” potentially harmful applications, such as virtualization, are much less mature on mobile devices than they are on PCs.

These unique challenges suggest that traditional malware analysis and detection methods need to be rethought in the context of mobile devices. Currently, network-based identifiers (e.g., network traffic patterns) are considerably more actionable for mobile device than host-based identifiers (e.g., writing a specific file). Indeed, a carrier or oper-

[1] Portions of this chapter previously appeared in [226] and [229].

ator could easily disconnect, and potentially reset, a mobile device that produces suspicious network traffic. On the other hand, detecting, on the device itself, that an application is malicious is much more complex without elevated privileges. In other words, given the current administrative models, network-based intrusion detection systems appear considerably more useful to mobile devices than their host-based counterparts.

We use these insights to propose "A5," short for Automated Analysis of Adversarial Android Applications. A5 is a system that draws conceptual design from existing dynamic analysis (or "sandbox") systems. Systems like A5 can be employed by market proprietors as a means of policing their marketplace. Likewise, entities such as enterprises, academic institutions or cellular providers may employ A5 to better protect their users.

At a high level, A5 executes malware in a sandbox environment that consists of some combination of physical devices and virtual Android systems hosted on a PC. A5 allows malware to connect to the Internet, in order to record network threat indicators and create network IDS signatures. These signatures can in turn be used directly by an enterprise to protect mobile devices that connect to the Internet through a corporate network or to protect all corporate devices by forcing mobile device traffic through a network proxy. Similarly, cellular providers could use these signatures to protect devices connected to carrier networks.

The key novelty in A5 is to use a combination of static and dynamic analysis to *coerce* the application into triggering its malicious behavior. Indeed, in mobile applications, activity can be triggered by a wide assortment of system events—for instance, receiving a phone call, or having the screen go into lock mode. A5 attempts to exhaustively determine all possible paths that can trigger malicious behavior, before separately evaluating them. Doing so, A5 can capture activity that would be missed by naïvely executing the malware (i.e., simply "clicking on the icon"). Furthermore, by combining physical devices with virtual Android images, A5 can capture a wider range of malicious behavior than a sandbox solely based on emulation would and can correctly process malware that employs certain types of sandbox evasion techniques. Likewise, A5 can accommodate a wide range of different hardware and software (e.g., SDK) configurations. To the best of our knowledge, A5 represents the first open source sandbox designed for mobile malware. The source code for A5 can be found at `https://github.com/tvidas/a5`.

In the remainder of this chapter, we first introduce background on static and dynamic analysis in section 5.1, where we also differentiate A5 from the relatively large body of related work on Android security. We then describe the design and architecture of A5 in section 5.2. We present a performance evaluation of our current implementation of A5 in section 5.3, notably showing that, using parallelism, A5 is able to analyze 1,260 unique malware samples in just over 10 hours using commodity desktop hardware. We detail specific limitations of A5 in section 5.4 and we consider the general limitation of sandbox analysis evasion in section 5.5. We draw conclusions in section 5.6.

5.1 Background and Related Work

Without access to source code for analysis, inspection and understanding, one must resort to other techniques when analyzing compiled software. In the context of malware analysis, *dynamic analysis* involves executing the malware samples to observe their behavior [188]. Conversely, *static analysis* refers to techniques that inspect or process a sample, but never execute the malware [107]. Manual, static analysis, colloquially known as as "reverse engineering," can be very effective, but often requires highly trained individuals and is time consuming. Thus, it is difficult to scale manual analysis at the pace that mobile malware is growing—both in terms of volume and in number of existing variants [106, 109, 147, 225, 248].

A dynamic analysis technique often used in vulnerability discovery can be automated to process input to samples automatically. *Fuzzing* is the process of sending data as input to a program, possibly intentionally invalid data, in order to coerce a desired condition or behavior. The input can be created programmatically to cover a range of inputs, and in this way can be thought of as a brute-force attack against the software. This technique may be considered inelegant, but fuzzing implementations are often straight-forward, and effective. Fuzzing is used in automated vulnerability discovery to find software vulnerabilities that are not feasible to audit in any other way [213].

Malware sandboxes. Malware sandboxes automate dynamic analysis techniques to inspect large volumes of malware automatically. The general operation of a sandbox system is to execute each input sample much like a user would, but in a controlled environment instrumented to monitor host and network activity. The sheer volume of unique malware samples on traditional computers makes the use of automated sandboxes appealing. Numerous commercial products, such as CWSandbox [240], and academic projects, such as ANUBIS [63], have appeared over the past several years. Automated sandboxes often scale linearly with computational power. A sandbox addressing computer malware may boot a virtual machine, copy the samples to the virtual machine, then execute the sample. The sandbox can monitor and report on changes to the host (e.g., registry keys, files) and network communications. For instance, Rossow et al. present a dynamic analysis system called Sandnet [188], which is used to collect network traffic from PC malware samples. Sandnet is used to process 100,000 samples and the authors find that Domain Name System (DNS) and HTTP have novel trends in malware use.

Malware analysis systems for Android. The work most related to A5 is a dynamic analysis system called Andrubis [4]. Andrubis is an extension to the automated PC malware analysis project ANUBIS, but is designed for processing Android packages. The inner-workings of Andrubis are not publicly known, but the creators allow anyone to interact with a public interface via website. Blasing et al. [66] describe another dynamic analysis system for Android. Their system focuses on classifying input applications as malicious (or not). The system instruments Linux features and scans applications for the use of potentially dangerous criteria. Like Andrubis, this system interacts with the malware by starting the application's primary Android Activity.

A5 differs from these two systems primarily in the way A5 interacts with the malware—using multiple techniques to coerce the execution of the malicious code. However, as detailed in section 5.2, there are several other differences such as the parallel implementation of A5, support for every Android API version, and the ability to use virtual instances, physical devices or both.

DroidBox [138] is a generic app monitoring tool for Android apps. It monitors an Android app for various activities at runtime, such as incoming and outgoing network data, file read and write operations, services started, etc. It then provides a timeline view of the monitored activity from the app. DroidBox is useful for manually identifying malware by viewing its observed behavior. Compared to A5, DroidBox does not automatically coerce the app into undertaking particular behaviors, and A5 specifically captures network traffic for finding malicious network indicators. In addition, A5 uses static-analysis in addition to dynamic monitoring of the app to find coercion points automatically.

Similar to A5's bytecode static-analysis, ComDroid [81] performs static-analysis of decompiled bytecode of Android applications. ComDroid performs flow-sensitive, intra-procedural analysis to find Android "Intents" sent with weak or no permissions—but contrary to A5, ComDroid does not perform any dynamic analysis.

A5 currently only captures network traffic to aid in finding malicious network indicators. It may make sense to pair A5 with taint tracking systems such as TaintDroid [97] in order to track host-based malware indicators. For instance, Andrubis employs TaintDroid. However, it may take considerable effort to extend TaintDroid to support all SDK target versions and to work with a range of physical devices, as A5 does right now.

Automated signature creation. Automating the tedious and error-prone process of creating network IDS signatures is a well-researched topic but remains an open problem. As a representative example, Kim and Karp create an automated system called Autograph that generates signatures for TCP-based Internet worms [135]. Like many efforts at automatic signature creation, Autograph's detection mechanisms are particularly designed to address one type of malware, in this case worms. As such, Autograph's pre-filtering step that discerns unsuccessful TCP connections, is not particularly useful for identifying malicious Android application traffic. A different system called Honeycomb presents similarities to A5's desired goal of automatically creating IDS signatures. Kreibich and Crowcroft describe the system which collects traffic from a honeypot and subsequently creates network signatures [137]. Since the network traffic is captured from a honeypot, the traffic is assumed to be malicious (or at least suspicious). A5 similarly assumes that all input is malware, but due to the repackaging common in Android malware, malicious network traffic is likely to be mixed with benign traffic.

5.2 A5 Architecture

The immediate need for a system like A5 is driven by increasing volumes of mobile malware. However, the design of A5 is also directed by several criteria borrowing from the more mature field of PC-based dynamic analysis and the unique nature of today's mobile device ecosystem. Here we enumerate a list of desired features for such a system, and describe an implementation designed to meet these goals.

5.2.1 Objectives and Design

Autonomy and scalability. The system must be able to handle volumes of malware without user interaction. As with PC malware, mobile malware is now growing at a rate that makes manual, human analysis unfeasible (see section 3.3).

Evasion resistance. The system must be able to adapt to evasion advances in malware. As seen in the PC, mobile malware is increasing in sophistication. With the advent of automated malware processing, malware authors have already begun to include minor attempts to evade sandbox systems.

Mobile-specific interaction. The system must interact with malware in Android-specific ways. It is more difficult to solicit malicious behavior from current mobile malware, than traditional malware. Simply executing the malware (i.e., "clicking on the icon") may not exhibit any malicious behavior. Indeed, current mobile operating systems permit applications to register a software handler for a wide range of system events; for instance, receiving an SMS, screen going in lock mode, and so forth. Any such event may trigger some application code. Traditional computer programs may receive input along with execution; mobile applications may receive input along with a myriad of system events. In either case, the behavior may depend upon the input to the application.

Network-level indicator collection. The system should primarily collect network threat indicators. Host-based indicators, such as the modification of a file found on the device, are of limited value on Android. Indeed, since Android's architecture does not permit file system hooks, and, more generally does not even permit privileged access to most components of the system, it is not possible to implement controls similar to AV products found on the PC. Even if Android's architecture were adapted ex post facto to permit such products (e.g., by systematically rooting devices), network indicators are particularly useful to cellular carriers and/or wireless network operators to protect the device even without the ability to install controls on the device itself.

Modularity. The system should have a modular, expandable design. Mature analysis systems need to have interfaces allowing for the system to interact with other software systems such as IDS or firewall management tools. This requirement is generally driven by entities that have larger research and analysis environments of which A5 may

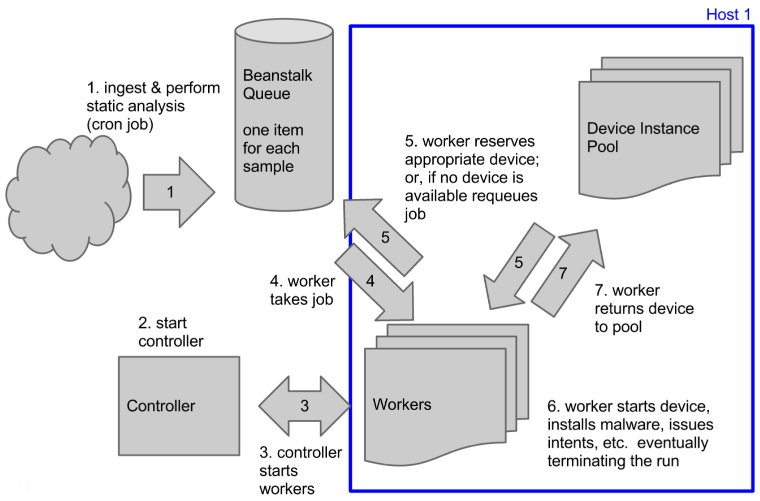

Host 1
1. ingest & perform static analysis (cron job)
Beanstalk Queue
one item for each sample
1
5. worker reserves appropriate device; or, if no device is available requeues job
Device Instance Pool
5
4
5
7
4. worker takes job
7. worker returns device to pool
2. start controller
Controller
3
Workers
6. worker starts device, installs malware, issues intents, etc. eventually terminating the run
3. controller starts workers

Figure 5.1: **A5 architecture.** Malware is first ingested (1) into a shared job queue. Independently, an overall controller is started (2) which starts one or more Worker processes (3). Each Worker retrieves jobs from the queue (4) and either postpones work or reserves a device instance (5) for dynamic analysis (6). Once analysis is complete, the device is returned to the ready pool (7). There may be many Workers and Device Pools on a single host. The controller and job queue may service many hosts (each with many Workers). Figure from [229].

become a component. Additionally, the system must be generally able to adapt to unforeseen circumstances, such as malware that exhibits some new behavior or technology that was not yet imagined when the system was designed.

Based on these objectives, we made the following design choices for the A5 architecture. To process as much malware as possible, A5 is highly parallel and distributed. The basic steps involved are shown in Figure 5.1 and detailed in the following sections. A5 consists of a queue, a main controller, and a set of workers which interact with a pool of device instances—these device instances are a combination of hardware resources (i.e., a specific phone model), and Android images running as virtual machines on a traditional PC.

First, malware is moved into the system and two stages of static analysis are performed to determine methods of interacting with the malware. Once static analysis is complete, an entry is created in a job queue for subsequent dynamic analysis. Later, one of many worker processes retrieves the job from the shared queue and executes the malware using an available device from the device pool. The dynamic analysis is informed from the static analysis; this combination of static and dynamic analysis allows our system to better coerce malware to execute nefarious behavior.

The remainder of this section details the implementation of A5. In particular, each of stage 1 static analysis, stage 2 static analysis, and dynamic analysis are detailed. Then, the concept of device pools consisting of virtual and physical devices is described, followed by a discussion of some Android-specific interaction techniques.

5.2.2 Malware Ingestion

A5 assumes all input samples are malware. The primary functions of the ingestion process are to create a shared job queue entry (we use beanstalkd[2]), to calculate several pieces of meta-data (such as cryptographic hash values), and to initiate the static analysis.

A5's ingest process is designed to run on each individual sample. This allows for on-demand submission, such as what one may expect of a web service, and as a batched process consuming samples periodically. In this way, all of A5 may run perpetually with no interaction from a user. Many security companies receive thousands of samples daily from sources such as VirusTotal [233] or MWCollect [239]. These incoming samples can easily be sorted, for example, to collect all Android samples in one location for input into A5.

5.2.3 Static Analysis

A5 first resorts to static analysis to try to detect potentially malicious actions. In Android, applications are usually written in Java (though at least 5% or as much as 70% contain "native" C components [232, 249]), and are distributed as APK files. These APK files are in fact Zip archives, which contain compiled Java classes (in Dalvik Executable Format (DEX) format), application resources, and an `AndroidManifest.xml` binary XML file containing application meta-data. The structure of Android applications and the Android security mechanisms have been well-documented [100, 203] and many tools exist for creating and manipulating APKs [43, 46].

Typically, Android applications that have a user interface specify at least one Android *Activity* and those that do not have a user interface specify at least one *Service*. These are classes that typically contain the core functionality of the mobile application, and are the primary method for executing application code. Much of the interaction with an Activity will be through the GUI. However, a Service may exhibit no GUI components at all, requiring different interaction during later dynamic analysis.

Android Inter-Process Communication (IPC) typically occurs in the form of an Android event known as an *Intent*. For instance, Intents are used to transfer information between applications and to notify applications when a particular system event, such as the receipt of a text message, has occurred. Since Android Services have no GUI, it is precisely these types of events that initiate a Service.

The chief output of static analysis is an enumeration of "interaction points" (e.g., Activities) and a set of "receivable intents" (e.g., `BOOT_COMPLETED`). Any of these may cause the application to take actions that would not normally occur if the application was simply launched using the graphical interface. As such, A5 will use these sets in order to coerce behavior from the malicious application. Many of these meet the need for better mobile-specific interaction.

[2]beanstalkd can be found at `http://kr.github.com/beanstalkd/` and is described as "a simple, fast work queue" originally designed to reduce latency on high-volume websites.

```
1  <receiver
2    android:name=".message.SmsReceiver"
3    android:enabled="true"
4    android:exported="true" >
5    <intent-filter
6      android:priority="214783648" >
7      <action
8        android:name="android.provider.Telephony.SMS_RECEIVED" >
9      </action>
10   </intent-filter>
11 </receiver>
```

Figure 5.2: **Receiver from ANDROID-DOS malware.** A5 automatically notes the action for this receiver. Receipt of a text message may be the only way this method is executed. In other words, the `.message.SmsReceiver` code may never be invoked if the instance never receives a text message.

Stage 1 Static Analysis: AndroidManifest

Much of the stage 1 analysis in A5 revolves around the `AndroidManifest.xml` file. This file dictates much of how an application may interact with the device. Each application must advertise the desire to receive particular Intents by declaring permissions in the AndroidManifest. Similarly, through documentation [227] and source code analysis [108], use of certain API functions implies the ability to receive certain Intents.

Even though the manifest is stored in binary XML form, tools are readily available for parsing key components such as requested permissions, broadcast receivers, background services, and activities. Each of these components define key interaction points for the application, and are cataloged for later use in dynamic analysis.

For instance, an Android `BroadcastReceiver` (or "receiver") is a way for an application to register the desire to receive an Intent from the system or another application. A receiver from recent Android malware is shown in Figure 5.2. A5 parses and saves the action from this portion of the manifest, in this case receipt of an SMS message. During dynamic analysis, the receipt of a text message may be the only action that invokes the `.message.SmsReceiver` method (Figure 5.2, line 2). Therefore, analysis systems that do not employ this SMS interaction will never encounter the malicious behavior.

Instead of creating yet-another-tool to extract pertinent information, we elected to leverage an existing open source tool known as Androguard [2]. If Androguard did not support a particular function that A5 required, we implemented the feature and submitted patches back to the Androguard developers.

Stage 2 Static Analysis: Bytecode

In addition to the relatively naïve stage 1 analysis of the Android application manifest, we also analyze the Java bytecode of the application binaries. The goal of this stage 2 static analysis is to identify additional interaction points

```
1  public static void h(Context paramContext)
2  {
3    IntentFilter localIntentFilter = new IntentFilter();
4    localIntentFilter.addAction("android.intent.action.USER_PRESENT");
5    localIntentFilter.addAction("android.intent.action.SCREEN_OFF");
6    paramContext.registerReceiver(new UserActivityReceiver(), localIntentFilter);
7    return;
8  }
```

Figure 5.3: **GoldDream malware example.** Code section reverse engineered from a GoldDream malware sample. Here, the desire to receive USER_PRESENT and SCREEN_OFF Intents are registered dynamically—these Intents do not appear in the AndroidManifest.xml and would be missed by analysis techniques that do not incorporate bytecode level analysis.

which enable users or the system to interact with the application. While many interaction points are declared in the application manifest, some may be created dynamically by the application, thus being missed by naïve analysis.

An example of an interaction point that may be missed during stage 1 is shown in Figure 5.3. The application in Figure 5.3 performs a `registerReceiver` call registering the desire to be notified when either the user begins interacting with the device or the device screen turns off. Neither of these Intents are found in the `AndroidManifest.xml`.

The stage 2 static analysis algorithm is fairly intuitive. First, A5 invokes the ded [98] decompiler to create Java classes from the Android application code. Next, A5 uses Soot [221] to obtain an Intermediate Representation (IR) and a Control-Flow Graph (CFG). This abstract IR is known as Jimple [222] and is useful because it eases the burden of dealing with the more complex Java bytecode.

Each node in the CFG represents one Java statement, and the graph edges correspond to the relationship between the statements in the malware. A5 traverses the CFG in order to find nodes that represent a known Android interaction point.

Each CFG node is further decomposed into an AST representing individual components of the statement. Specifically, A5 looks for calls to `android.content.Context.registerReceiver()` and `android.app.Activity.startActivityForResult()`. Calls to `android.content.Context.registerReceiver()`, as shown in line 6 of Figure 5.3, result in the application becoming eligible to receive Intents with a specified Action (i.e., a particular string). Similarly, calls to `android.app.Activity.startActivityForResult()` result in the application making a call to another application, but with an embedded Intent for the callee to make a callback to the target application. When A5 discovers one of these calls, the CFG is recursively traversed in order to resolve variable definitions. These definitions must be resolved in order to capture Intents that represent interaction points. For example, Figure 5.4 shows a call to `registerReceiver` at line 5,

```
1   Calendar c = Calendar.getInstance();
2   String bootc = "android.intent.action.BOOT_COMPLETED";
3   int seconds = c.get(Calendar.SECOND);
4   intentFilter bootcif = new intentFilter(bootc);
5   registerReceiver(bootcif);
```

Figure 5.4: **Variable resolution example**. Code section demonstrating the need to resolve variables. In this case the CFG is recursively traversed in order to find the value of bootcif at the time `registerReceiver` is called in line 5. In this case, A5 concludes that the program dynamically registered the desire to be notified when the system has finished booting.

however, the AST node only contains the component for variable `bootcif`. A5 recursively traverses the CFG to determine the variable definition at line 2.

Much like the stage 1 static analysis, the output of the bytecode static-analysis is a set of receivable Intents for use during the dynamic analysis.

5.2.4 Dynamic Analysis

The ingestion process and static analysis components execute relatively quickly, but the dynamic analysis portion is more time-consuming. Fortunately, it also lends itself to parallel execution. Figure 5.1 depicts many workers on a single A5 host. *Worker* processes on each A5 host retrieve jobs from the shared job queue for processing. Once a new job has been reserved from the queue, the Worker inspects a pool of candidate Android instances available to that particular host attempting to reserve a compatible instance. A compatible instance is one in which the malware sample is expected to run. For example, a mobile application that declares a minimum SDK (Android API) of 8, will not run on a Android API 4 device. Even if the application were to be modified by A5 to specify API 4 prior to instance selection, the application may actually rely upon a feature not available until API 8. Assuming a compatible instance is available, the Worker continues with dynamic analysis. If no compatible device is available, the job is placed back into the queue with a delay in order to reduce the chances that Workers repeatedly reserve the same job when no compatible instance is available—effectively de-prioritizing other pending samples.

The Worker then boots the instance and follows the process depicted in Figure 5.5. Communication with a running instance is performed with the SDK debugging tool known as the ADB. Since the boot may take some time, the worker initiates the boot process, then, using ADB, blocks until the device is fully booted. Once booted, A5 uses ADB to install the malware sample into the instance. Once the sample is installed, the Worker coerces malicious behavior from the instance, again using ADB. After a configurable period of time the Worker terminates the instance and returns it to a known state.

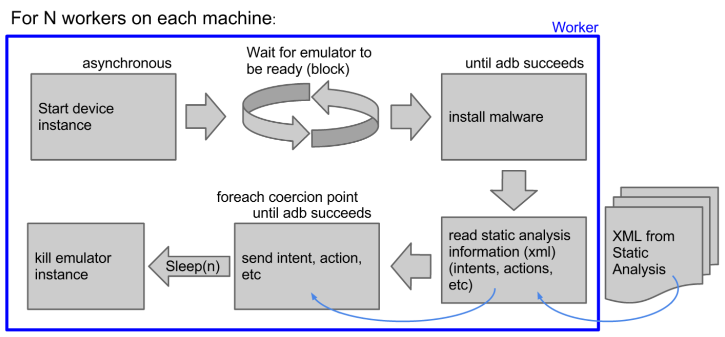

For N workers on each machine:
Worker
asynchronous
Start device instance
Wait for emulator to be ready (block)
until adb succeeds
install malware
foreach coercion point until adb succeeds
read static analysis information (xml) (intents, actions, etc)
XML from Static Analysis
send intent, action, etc
Sleep(n)
kill emulator instance

Figure 5.5: **A5 Worker process flow.** Communication with the device instance is performed using the ADB, and output from the static analysis is utilized in dynamic analysis interaction. Figure from [229].

5.2.5 Instance Pools

When retrieving a new job, the Worker must locate a device instance compatible with the malware sample. For this reason, A5 maintains pools of devices on each host. Each instance in the pool may be a physical or virtual instance, as detailed below. Workers synchronize the use of instances by maintaining instance state in a data structure shared among Workers on each host.

Virtual instances have the benefits of being low-cost, easy to automate and generally flexible. On the other hand, virtual devices are not typically used in everyday computing, so malware that can detect the virtual environment may elect to exhibit alternate behavior. In this light, the use of physical devices may be desirable.

Virtual Instances

Virtual instances in A5 are realized with modified versions of the emulator distributed with the Android SDK. These instances all are stored and executed on commodity computer hardware. Using virtual instances allows A5 to scale easily by simply creating larger instance pools as needed. Resetting a virtual instance to a known state is as simple as starting the instance with a “wipe data” flag or deleting and recreating the instance image from scratch.

However the emulator offers a subset of features found on a real device. The lack of features can be coarsely grouped into two classes: features not implemented by the emulation system and software that is not present in the emulated device image. An example of the former is Bluetooth, which is not implemented in the emulator, but is present on most devices. An example of the latter is the Google Play application, which is pre-installed in nearly every Android device sold, but is not present in the default emulated device image. The lack of features presents a fundamental problem to a dynamic analysis system: if malware makes use of one these missing features, the malware will not run properly. This change in behavior may be explicit (malware employs “virtualization detection”) or implicit (the malware happens to try to use a missing feature). In either case, the desired behavior from the sample will not be realized. For this reason, evasions are considered in-depth in section 5.4.

```
String v0 = TelephonyUtils.getImsi(((Context)this));
if(v0 == null) {
   return;
}

public static boolean isEmulator() {
   boolean v0;
   if((Build.MODEL.equalsIgnoreCase("sdk")) ||
       (Build.MODEL.equalsIgnoreCase("google_sdk"))) {
   v0 = true;
   }
   else {
      v0 = false;
   }
   return v0;
}
```

Figure 5.6: **Android.hehe emulator evasions.** Simple evasions are starting to appear in real malware observed "in-the-wild." This example is from Android.hehe malware, which attempts to evade analysis by detecting that an instance is an emulator via Build strings common in developer SDKs, and by checking for the lack of a device subscriber ID which is common in the Android emulator.

On traditional computers, virtualization detection was initially not found at all. It was later employed more frequently to evade dynamic analysis systems: If the malware detected it was running within a virtual machine, the malware would demonstrate benign behavior. The increased use of virtualization detection by malware in turn led to creating dynamic systems employing physical machines [209]. However, as virtual machines and, more generally, virtualization, is increasingly employed on laptops and desktop computers, running in a virtual machine is not a give-away that the platform is attempting to analyze a piece of malware. In other words, new forms of malware may paradoxically employ virtualization detection *less* frequently.

In today's mobile computing paradigm, the devices physically move with the user and data bandwidth at any given time varies greatly. Contrary to traditional computing platforms, there is not yet a clear mobile use-case for virtualization in typical end-user environments. Resources such as bandwidth or power are far more constrained and devices are typically not shared among multiple users. However, if systems like A5 are increasingly deployed—as we believe they will be—virtualization detection in mobile malware will become a reality.

Manual analysis of malware families in 2012 [248] revealed that the current generation of mobile malware did not yet employ virtualization detection. Even so, our previous work explored possible methods such detection might be implemented and provided a taxonomy of several methods [226]. Following this work, recent malware is starting to employ such detections "in-the-wild". For instance, a recent Android Malware (Jan 21, 2014), Android.hehe [86], implements two checks: (1) the nonexistence of an International Mobile Subscriber Identity (IMSI)—a unique cellular subscriber number and (2) the existence of `Build` strings that are exactly "sdk" or "google_sdk" as shown in Figure 5.6. Similarly, Android Malware "Oldboot" (Apr 2, 2014) identifies the running location of the malware instance

(`/sbin/meta_chk`) and exits if the path is not as expected or if there is no Subscriber Identity Module (SIM) card present.

In A5, the emulator software is built from source, and subsequently the resulting emulator is similar to the emulator distributed with the binary Android SDKs. However, we enhanced the emulator to evade some virtualization detection features. For example, an unmodified emulator will always return the same values for APK calls such as `TelephonyManager.getDeviceId()` (all zero's) or `Settings.Secure.ANDROID_ID` (null). By modifying the virtual instances such that values indicating a physical rather than a virtual device is in use, A5 becomes less detectable by malware seeking to determine if execution is occurring in a virtual sandbox. A5 makes such changes in `Build` parameters, the `TelephonyManager` class, and the default networking configuration. A5's emulator instances have configurable settings in these cases each instance returns values that simulate values observed on real devices.

However, even with modifications that make the emulated Android instance more like a physical device, there are large voids in the emulated environment. Malware need only check for one of the many hardware features not currently implemented such as Bluetooth, WIFI, sensors, etc. These hardware features are very common in physical devices and are simply not present in the emulator. Implementing entire systems to emulate these is a large undertaking and has not been done as part of the current A5 implementation. For this reason, it is currently trivial for malware to detect that the malicious application is currently running in a virtual environment and not a real device. To address this issue, we complement our virtual environment with physical device pools.

Physical Instances

By using real, physical devices in A5 device pools, we prevent malware from being able to trivially determine from hardware presence that the sample is being processed by a dynamic analysis system. The real devices possess actual hardware for systems that are missing in the virtualized environment, such as Bluetooth. A5 systems relying upon physical instances can scale linearly by purchasing more devices.

Physical devices embody a wide range of software features and hardware capabilities. As with typical computers, more recent devices have more computing power and are distributed with more recent software. In order to process all samples with physical devices, at least one device is needed for every Android SDK target version.

Resetting a physical device to a known state is not as simple as resetting a virtual instance. Android devices do not typically have boot modes that can be controlled remotely, such as Preboot Execution Environment (PXE) found on many modern network adapters. In fact, typically the only boot-time interface to Android devices is the USB/charging port. Various manufacturers support proprietary flashing protocols, but some devices employ the more approachable *fastboot* protocol. Critically for A5, fastboot supports erasing from and writing to device data partitions. A device may enter fastboot mode prior to loading the operating system when a person uses a special hardware key combination. However, ADB can also be used to reboot a running device directly into fastboot mode. In this way A5 can programmatically return a physical device to known state prior to each execution. In order to write data to a partition, the device must be "unlocked." Manufacturers each hold different positions on whether a consumer should have the ability to write data to a device. Developer-friendly devices, such as the Nexus-branded devices, can all easily be unlocked. Similarly, these devices also support fastboot making them ideal devices for automated use in A5.

Unlike virtual devices, physical devices are capable of actual physical medium communication. Physical devices can communicate over cellular, WIFI, Bluetooth, NFC, etc. To permit devices to exhibit network behavior, A5 devices are configured to connect to a wireless access point that routes to the Internet. This wireless connection provides a method for network package capture, but can also leak sensitive information or be used by a miscreant to attack a local resource. Furthermore, the wireless environment around the device may change due to outside circumstances. For example, a neighbor may install a new, competing wireless access point. For all of these reasons, physical device pools should be placed into an Radio Frequency (RF) isolated environment along with the routing access point.

5.2.6 Malware Interaction

Regardless of the instance type, once the malware is installed, A5 must interact with the instance to coerce the malicious behavior. Of course, A5 can, and does, start the main Activity of applications that place an icon in the application list for users to click. However, A5 employs other techniques for interaction.

The primary method for interaction is via receivable Intents ascertained during stage 1 and stage 2 static analysis as described in section 5.2.3. Using ADB, a Worker can send Intents to a running instance. Intents are sent to the device sequentially and feedback from ADB can be used to verify that an Intent was successfully registered on the instance.

A5 could simply send every type of Intent available for the SDK version of the instance. However, this coarse style of fuzzing presents two problems: inability to discern custom Intents and poor performance. Each version of the SDK specifies dozens of Intents and permissions [44], simply iterating through all of these takes considerable time which would lower the overall performance of A5. More importantly, applications can specify custom Intents [44], which may not be known a priori. Figure 5.7 depicts the creation and receipt of custom Intent, `some.custom.intent.FOO`. For these reasons, A5 employs a more granular system, precisely issuing Intents derived from static analysis.

Some Intents do not require any additional information. Sending the `BOOT_COMPLETED` Intent unambiguously indicates that the device has finished its startup procedures. Other Intents lose meaning without additional information. For instance, consider a text message (SMS), which requires associated telephone numbers and message content. In order to handle these more complex situations, A5 uses a custom library, libIntent, to create Intents that consist of well-

```
1   Intent i = new Intent();
2   i.setAction(some.custom.intent.FOO);
3   context.sendBroadcast(i);
4
5   public class IncomingReceiver extends BroadcastReceiver {
6     public void onReceive(Context context, Intent intent) {
7       if (intent.getAction().equals(some.custom.intent.FOO)) {
8         //some action
9       }
10    }
11  }
```

Figure 5.7: **Custom Intent example.** A custom Intent is defined on line 2. A broadcastReceiver as defined in lines 5-11 can receive such an Intent by observing the custom string in the Intent's action (line 7).

formed data. Continuing with the SMS example, when static analysis has indicated that an application may receive an SMS, A5 uses libIntent to send text messages via ADB to the device with random message content and random, valid 10-digit telephone numbers as the source. In A5's current implementation, libIntent handles SMS messages, power events, battery level changes, network events (delay, speed, cellular type, etc), GPS data, Voice calls, and the ability to pass through generic events in cases where users wish to use libIntent independently from A5.

The final method of interaction is via a software feature known as a "monkey." Android's developer software distribution contains a program named `monkey` for use in user interface testing. The `monkey` program "generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events" [44]. By using the `monkey`, A5 can automatically simulate user input. This rather coarse method of graphically fuzzing the application can change the state of the installed program, possibly making the instance more likely to respond to interaction. For instance, consider an application that blocks access using a modal dialog with a EULA. A `monkey` may be invoked to "click" many times, eventually bypassing the access restriction.

5.2.7 Network Traffic

Any network communication destined for the Internet from an instance is routed to the Internet by A5. This is a design decision that permits malware that connects to remote servers to communicate with these servers so that the network traffic can be captured and used to inform network countermeasures. Other commercial and academic sandbox systems operate in a similar fashion. Some systems selectively route some traffic to the Internet and direct other traffic to controlled servers or honeypots [63].

For virtual instances, A5 utilizes the network capture feature of the emulator. Using the emulator feature is not only convenient to initiate, but also results in a single data file for each input sample.

For physical instances, the devices are connected to a WIFI network which is monitored. Unlike the emulator feature, network capture is performed on a shared medium. To obtain device-specific data, the network capture is filtered using the device's Media Access Control (MAC) address.

Regardless of the instance type, the resulting capture file will contain a combination of malicious traffic, administrative traffic (such as ADB), and, in the case of repackaged applications, legitimate traffic. Each of which presents a unique problem with regard to the creation of network countermeasures. The administrative traffic is generally very easy to filter. For example, ADB traffic can typically be filtered based on the TCP port. Since the ADB ports vary by the instantiation of each instance, it is unlikely that a TCP port-based filter will omit any data erroneously. This simple filter can reduce the network capture substantially. For instance, filtering ADB traffic reduces the network capture for the "NotCompatible" piece of malware [189] by 65%.

Unfortunately, filtering legitimate traffic is more difficult as the characteristics of the legitimate traffic are not known a priori. Repackaging of applications in order to add malicious functionality creates additional difficulty. One might imagine learning algorithms that observe similar behavior across a family of malware. However, with repackaging, the shared behavior may be the legitimate traffic from the original application or it may indicate the same malicious software was injected into several different applications.

In the current implementation of A5, the network captures are pruned such that much of the traffic that is not beneficial to network countermeasures is filtered out. From the network captures, analysts can then, for instance, create IDS signatures; while the process is not entirely automated, A5 facilitates this through an extensible plugin framework.

5.2.8 Plugin Framework

A5 further assists the analyst through a modular post-analysis framework. Plugins are Python scripts that conform to a simple API, requiring each plugin to only define a handful of functions and follow a naming convention. Each plugin is loaded dynamically by A5 during malware execution and evaluated against dynamic analysis results or submitted samples, or both. Plugins generate two types of output, signatures intended to be used in an IDS, and freeform text that is intended to provide additional context to an A5 report. An analyst can elect to deploy any suggested signatures in light of the A5 report including any information added by plugins.

The default plugins create candidate signatures blindly—that is, without having any prior knowledge of the malware. For example, malware may issue a DNS query in order to determine the IP address of a C2 server. A perfectly acceptable IDS signature may scan for this particular DNS query and block the communication, thus preventing the malware from further interaction with the remote controller. On the other hand, a DNS query may be legitimate traffic issues with the purpose of determining the IP address of a server that is used to save high score data in a game. Default plugins are a simple attempt to automate the analyst's work when a sample has never before been encountered by A5.

```
1  alert tcp any any -> any any (msg:"autocreated dns->host rule";
       content:"Host|3A20|"; content:"notcompatibleapp.eu.|0D 0A|"; within:64;)
2  alert udp any any -> any 53 (msg:"autocreated dns->dns rule";
       content:"notcompatibleapp.eu."; nocase;)
3  alert udp any any -> any 53 (msg:"autocreated NotCompatible data decryptor";
       content:"notcompatibleapp.eu"; nocase;)
4  alert udp any any -> any 53 (msg:"autocreated NotCompatible data decryptor";
       content:"3na3budet9.ru"; nocase;)
5  alert tcp any any -> any 8014 (msg:"autocreated NotCompatible data decryptor";
       flow:established, to_server; dsize:13; content:"|04|"; depth:1; content:"|01 05
       00 00 00 00 07 00|";)
```

Figure 5.8: **NotCompatible candidate Snort signatures**. Signatures are automatically created by A5's post-analysis framework following the analysis of a NotCompatible sample. Lines 1 and 2 were created by default plugins, whereas lines 3–5 were created by a custom plugin designed specifically for the NotCompatible malware family.

Since the default plugins are fairly generic, they may have limited value, but they do automate the signature creation process and may help prevent costly typographical mistakes in signature creation.

Unlike newly encountered malware, the plugin framework can also be employed to automate signature creation for known malware families. The framework facilitates programmatic access to the data collected during dynamic analysis and the associated Android application. In this way, malware can be identified using the submitted application, or via captured traffic, or both. Subsequently, prior knowledge can be applied to new variants of malware to automate many analysis tasks.

As we will show in section 5.3, a plugin can be used to automate specific identification, data extraction, decryption, and the crafting of IDS rules well suited for a malware family. A5 plugins provide a general method of applying actions that may otherwise take considerable time for an analyst.

We provide an example of candidate signatures created by the post-analysis framework in Figure 5.8. While the plugin here creates Snort signatures, other IDS (e.g., Bro) could be supported in a similar manner.

The signatures have been automatically created from the dynamic analysis of the NotCompatible malware [189]. An analyst would still be charged with identifying that the candidate on line 1 is likely of little value, but that lines 2–5 may be useful. Lines 1 and 2 were created by default plugins that assume that DNS queries still present in the pruned network traces search for malicious domains, and thus a DNS based rule or an HTTP rule based on the DNS name may be useful. However, the NotCompatible malware does not use the HTTP protocol, so a signature looking for a Host header is not relevant.

The real power of the plugin framework is actually more evident in lines 3–5. Rather than a specific piece of malware, NotCompatible actually defines a family of related pieces of malware—we have observed dozens of different samples in the wild. Using a previously seen variant of NotCompatible, a family-specific plugin was able to produce a custom rule for this (unseen before) variant of NotCompatible in lines 3–5. In other words, our plugin is able to automatically generate IDS signatures for new variants of NotCompatible as long as the overall networking protocol used in the malware family does not change.

More precisely, the plugin identifies NotCompatible malware based on the unique networking protocol employed by the malware. Once the malware is identified, the plugin can then automatically decrypt the C2 server data stored in the APK. This is very useful for the analyst since the dynamic analysis system is likely to only create network traffic to the initial command-and-control server, therefore never having to resort to the backup server.

5.3 Evaluation

Here we evaluate an implementation of A5. We evaluate our design goals with the specific implementation, measure performance of the system, and compare A5 with an existing Android runtime analysis system.

5.3.1 Design Goals

In section 5.2.1, we enumerated five criteria that our A5 must embody in order to be a successful sandbox. Here we discuss how A5 fulfills each of these metrics in turn.

Autonomy and scalability. A5's distributed and expandable architecture permit A5 to grow linearly with the growth of mobile malware samples. Additional Workers can be added to an A5 installation by adding resources to an existing node or by adding additional nodes. Further, the periodic ingestion process along with on-demand web service submission affords A5 the flexibility of interacting with a user dynamically or autonomously processing malware feeds without user interaction.

Whether samples are processed automatically or on-demand, A5 attempts to ease the burden placed on the analyst by filtering out innocuous traffic and presenting candidate signatures for the remaining network activity. For identified malware families, additional post-processing can be performed in order to further automate analysis and countermeasure creation. Ultimately, human interaction is still required for final determination of network countermeasures, but this interaction is substantially less than if the analyst had to perform any analysis manually.

Mobile-specific interaction. The methods of malware coercion certainly vary between mobile and PC-centric malware. Here A5 employs a variety of mobile-specific coercion techniques. The most straight-forward interaction is actually running the mobile application. For most applications, the interaction is similar to that of PC-malware: starting the default Android Activity. However, through two stages of static analysis, A5 determines many types of coercion that would otherwise be omitted from a dynamic analysis system. Intents determined from static analysis are then issued in an Android instance dynamically, effecting the malicious behavior.

Evasion resistance. When interacting with malware, A5 takes steps to present plausible information to the application under evaluation. For instance, text messages appear to originate from random, valid numbers. By using plausible data and avoiding predictable data, A5 seeks to be less identifiable to malware that inspects the high level state of the device.

Knowing that virtualization detection has a mature presence in targeting other malware sandboxes, it is a foregone conclusion that Android malware will employ similar tactics. Therefore, virtual instances used in A5 employ a modified Android emulator that takes steps to hide its virtual nature by embodying programmatic characteristics of a physical device. Even so, some types of virtualization detection are simply too difficult to implement as a modification to the standard Android emulator. A5 allows system implementers to mitigate this risk by using physical instances thus avoiding use of an emulator altogether. By using real devices, the malware actually runs on hardware making sandbox detection much more difficult for malware authors.

Unlike other Android sandboxes, A5 avoids a type of evasion in that malware targeting any particular Android API version will run in A5. If A5 only supported a subset of API versions then malware might not run in A5 at all, rendering no analysis and effectively evading the entire system. By allowing device pools to cover the entire range of Android APIs, among any combination of virtual and physical instances, A5 can start a Worker for any API version required by a particular sample.

Network-level indicator collection. The collection of network indicators is critical to enabling enterprises and network providers to protect their networks, even when individual mobile devices may have little or no security-oriented software. For virtual instances, A5 makes use of a feature built-in to the Android emulator to collect network activity during dynamic analysis. For physical instances, A5 makes use of commodity network capture tools used on a shared WIFI access point. Network traffic capture on the shared network is filtered based on the known MAC addresses of the physical devices installed in the A5 instance. For either instance type, A5 creates candidate network countermeasures for never before seen samples and performs extended post-analysis on recognizable malware families leading to additional network countermeasures.

Modularity A5 is written in a modular, object-oriented way. Interaction with dependent software (such as ADB) are written as libraries providing a re-usable interface. Use of support systems such as database and job queuing are also done in very standard ways using mature subsystems. In these ways, A5 is adaptable to existing research environments that have different workflows or special needs.

However, for the general user of A5, the more useful form of modularity comes from the post-analysis plugin framework. This framework facilitates in automating tasks upon dynamic analysis results and submitted malware

```
12:14:56.918698 IP 10.0.2.15.22219 > 10.0.2.3.domain: 22666+ A?
    notcompatibleapp.eu. (37)
0x0000: 4500 0041 a717 4000 4011 7b83 0a00 020f
0x0010: 0a00 0203 56cb 0035 002d 7610 588a 0100
0x0020: 0001 0000 0000 0000 106e 6f74 636f 6d70
0x0030: 6174 6962 6c65 6170 7002 6575 0000 0100
0x0040: 0120
```

Figure 5.9: **NotCompatible network traffic.** ASCII representation of network traffic captured by A5 from NotCompatible malware. NotCompatible's namesake domain is clearly present (line 1) in this DNS query from 10.0.2.15, an A5 device.

A5 Stage	**Min Time**	**Max Time**	**Average Time**
Stage 1 Static Analysis	1.9	5.6	2.6
Stage 2 Static Analysis	10.0	120.7	12.7
Dynamic Analysis	124.5	202.3	133.8
Total runtime	136.4	328.6	149.1

Table 5.1: **A5 runtimes per sample** (in seconds). Each time is was calculated by processing the entire dataset described by Zhou and Jiang [248].

samples. Plugins can be used to automatically create candidate network signatures, decode sections of network traffic, decrypt malware payloads, and any number of other uses.

Plugins currently implemented in A5 focus on Snort [186] IDS format, though modifying for other formats is often merely a matter of syntax. Since any given user may prefer a particular IDS, each of which may have very custom syntax, preprocessors, configurations, etc. the default plugins are likely of limited value to any given user. However, the modularity of the plugin framework allows simple creation of new plugins such that candidate rules can be readily deployed to a particular system.

We measured A5 performance using 1260 samples from the dataset described by Zhou and Jiang [248]. The A5 system is on an 8-core (Intel Xeon E5620 @ 2.4 GHz) Linux machine with 64GB of RAM. A5 was configured to use five Workers, and the arbitrary sleep time for each sample (see Figure 5.5) was set to 60 seconds. The entire device pool was set to be virtual instances, with two device instances for each SDK version. As shown in Table 5.1, the average runtime for samples was 149.1 seconds, requiring just over 50 hours of cumulative execution to run all 1260 samples [248]. By using five Workers, the time required to process the entire set was reduced to just 10 hours.

Stage 1 Static Analysis performed correctly for all samples. Stage 2 Static Analysis prematurely terminated on 10% of samples. In particular, the ded decompiler [98] failed to decompile 10% (131) of samples. Of the 1129 applications that successfully decompiled, Stage 2 Analysis discovered Intents in 4.5% (51) of samples (14 of 50 families).

We also analyzed some more recent malware using A5 and verified the results manually. For example, NotCompatible is Android malware that does not root the infected device, but nonetheless is able to act as a TCP proxy, tunneling malicious traffic through an infected device (see section 3.4.2). This malware has typical botnet like features such as the ability for a remote actor to control the software through a publicly accessible C2 server. A5 was able to identify remote servers associated with NotCompatible malware. The primary remote server was immediately observable in the DNS request (Figure 5.9) and subsequent malware communication. Manual static analysis also revealed the remote address, but required substantial time to understand the Dalvik classes and required decrypting a data file. One operation of the NotCompatible malware allows a remote server to update the remote destination address, so the next remote communication would interact with a different server. Static analysis would not be able to discern this next server address.

The NotCompatible malware also highlights the usefulness of A5's Android-specific coercion techniques. NotCompatible malware does not execute upon installation. Instead, the malware waits for certain system events, namely `BOOT_COMPLETED` or `USER_PRESENT`. In other words, NotCompatible's malicious service will only start when the device is rebooted, or when the user unlocks the device's screen lock. A5 is able to determine the necessity to mimic these events and successfully coerce the malware.

The post-analysis plugin is useful to create the candidate signatures (shown earlier in Figure 5.8). In this case, previous analysis was leveraged to create a plugin capable of automatically determining that, in addition to the traffic observed to the primary server, `notcompatibleapp.eu`, other infected devices may attempt to connect to the backup domain `3na3budet9.ru`, which is encrypted and stored within the malware. This specific sample was configured to use TCP port 8014 on both servers, which A5 was able to automatically detect, and create IDS rules, based on the generic template created from a higher-level description of the malware family.

5.3.2 Existing sandbox comparison

In order to highlight some of A5's advancements we can evaluate some metrics with another Android sandbox. Unfortunately the source code and design of other sandboxes are not publicly available, so for comparison, we submitted samples to Andrubis via its web interface [4]. We observe that when a sample is submitted, the estimated job completion time is always reported to be about eight minutes. If we submit samples very quickly, multiple submissions queue sequentially causing a delay before the submitted sample is processed, indicating that Andrubis does not process samples in parallel. For this reason, we purposefully submitted samples one at a time, waiting for a current sample to complete before submitting another. Even though this is a public web service, we did not observe any other user of Andrubis during our testing.

We created an Android application that extracted virtualization information mentioned in 5.3.1. Namely, the Android Build strings and `TelephonyManager` identifies Andrubis' runtime software as an Android emulator. From this, we conclude that Andrubis does not appear to employ physical devices nor emulator detection mitigations.

We also submitted a second application that was identical to this first application except for an appended null byte in order to give the otherwise identical application a different file size and associated cryptographic hash. This was meant to ensure that submitted applications are actually executed by Andrubis and to observe similarity between two similar samples submissions. The reports were very similar even though execution time for the two samples were reported to be 227 seconds and 337 seconds respectively, so there is some variance in Andrubis' application runtime.

We then created 14 variations of the test Android application, each different only in that the settings for minimum, maximum and target SDK as configured in the `AndroidManifest.xml`. It seems that Andrubis only supports a subset of Android SDK versions supported by TaintDroid, namely 2.1 and 2.3. For example, submitting an APK with the minimum, maximum and target SDK each set to 8, denoting Android 2.2, in the AndroidManifest causes dynamic analysis to fail. Andrubis fails in this way for many combinations of SDK values causing the sample to not be analyzed at all.

Another difference is that Andrubis does not appear to use the static analysis portions to inform the dynamic analysis. This is easily evidenced by submitting an application that takes no action except upon some other, outside event, such as the receipt of a text message, or activation of the screen lock. Unfortunately, this means that for malware like NotCompatible, Andrubis will not observe any of the malicious behavior.

In practice, Andrubis often does not start processing samples immediately, in fact, only once did a sample start processing immediately. Instead, of the 15 submissions, Andrubis reported an average delay of 92 minutes (min: immediate, max: 16 hours). Overall, the execution time across our submissions averaged 253 seconds (min:227, max:337 seconds). Given that both Andrubis and A5 must incorporate certain delay in dynamic processing, the difference in average runtime is not significant.

As a final difference, A5 reports contain only meta information about the sample, but also contain IDS signature candidates. In this way A5 provides immediately actionable results. Conversely, Andrubis reports do not contain IDS signatures and may require more evaluation from an analyst, who then must manually create IDS signatures.

5.4 A5-Specific Limitations

In section 5.2, we detailed several components of our analysis system, A5. Design decisions and implementation choices affect the capabilities of each component of A5. In this section, we address limitations specific to the current state of our A5 implementation, analyzing both static (section 5.4.1) and dynamic (section 5.4.2) limitations. Then we turn to an analysis of the more general problem of analysis evasions in section 5.5.

As shown in section 3.3, malware is growing rapidly. Accordingly, one of A5's design goals was scalability. Currently, scaling A5 involves a compromise. Use of virtual instances enables simple, rapid scale at the cost of reduced resistance to evasion. On the other hand, use of physical instances enables much better resistance to evasion, but scaling physical instances requires labor-intensive procurement and management of physical devices.

Our implementation of A5 creates candidate IDS signatures. Since, most Android malware is repackaged (see section 3.3), there is a high likelihood that benign network traffic will be interleaved with malicious traffic. A5 may be extended to partially address this issue (see the plugin in Appendix 8.1), however, this remains an open problem.

As A5 is a hybrid static and dynamic analysis system, it shares many of the benefits of both types of analysis. However, A5 also inherits shortcomings of both types.

5.4.1 Static Analysis

Fundamentally, by virtue of static analysis being static, any dynamically received code updates by the application cannot be analyzed purely statically. For instance, the static analysis will have no access to bytecode or Java/Dalvik classes received at runtime and subsequently executed. However, A5's dynamic-analysis phase will still be able to collect network indicators of the dynamically executed code. This type of *update attack* [248] has been detected in malware in the wild.

Also, immediately of interest to any software system is sub-component dependency. In static analysis, A5 depends upon tools such as Androguard [2] and Soot [221]. Deficiencies in these tools may also manifest in A5. In some cases this is simply the standard software system engineering problem of ensuring component compatibility and over system stability. However, since A5 is a system designed to interact with malware, the security posture of each subcomponent (and A5 as a whole) is of particular interest.

The bytecode static analysis in A5 is limited to finding only behaviors that are statically defined and extractable from the bytecode itself. For instance, a given application component such as an Activity is started either by the Android middleware, or by another application, by sending an appropriate Intent to the application. Ideally, we would want to know all Intents receivable by the application, so we can coerce its behavior. Unfortunately, for methods which receive Intents, the method is provided the Intent via an `android.content.Intent` object, which contains a string with the name of the Intent used to trigger the target application. As the value in this string is filled in dynamically at runtime when the Intent is created, A5 is unable to statically determine the string content. In order to statically identify the Intent that triggered the activity or other application component, every Intent would need to be of a different type which is statically specified. However, this is not the case, and A5 is unable to extract or identify Intents received by receivers. Hence, the bytecode static analysis is able to extract only Intents which the application registers to receive, as the Intent action string is set statically when notifying the system of the Intents to receive.

A5's implementation of CFG traversal is intra-procedural, and the CFG analysis is flow-insensitive meaning all branches and loops are ignored. While this could result in unsound analysis, this has not been an issue in the context of A5. Indeed, the two types of interaction points currently sought are method invocations and interaction points are extracted by observing the values of arguments passed to the method. Typically, the argument construction leading to the method invocation would be linear code in the same basic block without any conditional statements. In this case, flow-insensitive analysis would be sufficient for analysis. Flow-insensitive analysis may also introduce false positives. However, since in the case of additional false positives the malware coercion would only require a subset of identified interactions which would result in effective coercion. The primary issue with false positives is decreased performance due to unnecessary coercion.

5.4.2 Dynamic Analysis

A5 exhibits well-known limitations similar to any dynamic analysis system. For example, if the malware behavior changes based on time of day, the success of A5 would depend greatly on when the sample happened to be selected by a Worker. Similarly, if malware targets a specific manufacturer other than the manufacturer employed in dynamic analysis, the malware may exhibit alternate behavior during analysis. Generally, it is simply not possible to ensure that all possible functionality of a sample is explored during execution [107]. As with any sandbox system, the primary use is to quickly process volumes of malware—handling most samples correctly, not focusing on individual accuracy of a given sample.

There is some evidence that randomly issuing input to the user interface may not yield successful results [118]. Gilbert et al. conclude that random input, as is expected from an Android Monkey, may yield as low as 40% coverage of all possible UI input. On the other hand, more structured input is explored by Zheng et al. [246] which exhibits its own drawbacks, in particular the inability to handle certain logic when interacting with the UI. This particular implementation [246] has other drawbacks, but many can likely be overcome. Much as the hybrid static and dynamic analysis employed by A5, the best UI interaction is likely a combination of random and structured input. Additionally, new methods of UI interaction are increasingly available as Android's SDK evolves. Devices using API 16 or higher have a "Dump view hierarchy" that could be used to inform a UI automator. This hierarchy presents a tree of UI components that could be traversed as part of dynamic analysis. Continuing with the previous example of the EULA modal dialog, using such a UI automator could enable a dynamic system to precisely "click" an accept button instead of randomly happening to "click" the screen in an acceptable location.

In practice, ADB is not reliable, often not performing the desired action, yet returning a successful return code. In order to perform in lieu of these faults, A5 monitors ADB execution repeating actions until the desired effect is observed.

5.5 Evasions

From a defender's perspective, when a new piece of malware is discovered, it must be analyzed in order to understand its capabilities and the threat it represents. Dynamic Analysis, a popular method of analysis employed by A5, consists of executing the malware in a controlled environment to observe effects to the host system and the network. This topic is well-studied for PC malware, however is relatively immature for mobile malware. For this reason we explore the topic deeply, exploring the relatively large set of related work in section 5.5.1, proposing several detection techniques across emulation (section 5.5.2), behavior (section 5.5.3), performance (section 5.5.4), hardware and software components (section 5.5.5), and system design (section 5.5.6). We further explorer the feasibility of these evasions with respect to Android permissions in section 5.5.7 and we evaluate the techniques against existing sandboxes in section 5.5.8.

On traditional PCs, the controlled environment used for dynamic analysis is often created through host virtualization. PC malware authors have consequently taken to design malware that can detect virtual environments [112, 139, 173, 190], and exhibit alternative, benign behavior, rather than the malicious behavior which would occur on actual hardware. This kind of evasion allows malware to thwart dynamic analysis systems, and has fueled an arms race between those improving the realism of virtualized environments and those wishing to detect them.

However, virtualization technology has considerably matured in the past few years, and a number of users have migrated physical machines to virtual instances. Today, production services may be run in corporate virtualization "farms" or even in space rented in a "cloud." As a result, virtualized environments are not merely used for sandboxing malware anymore, but have become commonplace for a number of applications. In turn, the ability for malware to detect a virtual environment is losing its usefulness, as virtualization is no longer a near-certain indication that dynamic malware analysis is taking place.

On the other hand, there are, so far, only limited use cases for virtual environments on mobile devices. Likewise, emulated environments are typically only used by developers or dynamic analysis systems. For this reason, mobile malware authors may still employ virtualization or emulation detection to alter behavior and ultimately evade analysis or identification.

In this section we contribute several techniques that can be used to detect a runtime analysis in Android. Some of these techniques are specific to the Android framework or a virtualized environment (e.g., emulator), and could be quite easily thwarted, while others, based for instance on resource availability, are more general, and much harder to defend against. All of the techniques we present require minimal or no privileges, and could be invoked from typical applications found in online marketplaces.

The primary contribution of this section is thus to demonstrate that dynamic analysis platforms for mobile malware that purely rely on emulation or virtualization face fundamental limitations that may make evasion possible.

5.5.1 Related Evasions Work

Automated runtime analysis systems are a popular method of processing large volumes of malware, or as an alternative to having skilled personnel perform lengthy manual analysis. For desktop operating systems there are numerous free [63] and commercial [240] systems that perform such analysis. More recently, a few systems [1, 4, 6, 26, 66, 138, 145] have been proposed for dynamic analysis of mobile malware. Because of their novelty, these systems remain less mature than their desktop counterparts.

PC malware writers responded to the advent of automated systems by finding creative solutions to detect whether code was running in virtualized environments. Virtualization detection routines have been implemented in numerous ways from detecting a vendor-specific API, such as VMWare's communication channel [139], to observing environmental effects of a single Central Processing Unit (CPU) instruction [190].

Upon detection, malware may exhibit alternate behavior, hiding its true purpose. This runtime change in control flow results in a different execution path through the malicious software. In 2008, Chen et al. observed that PC malware exhibited reduced behavior in nearly 40% of unique samples when run in debugged environment, and 4% of samples when run in a virtualized environment. However, the samples that exhibit alternate behavior due to environment accounted for 90% of attacks during particular time periods [78]. Moser et al. propose a system for analyzing multiple execution paths by presenting different input and environments to subsequent executions of the same malware [161]. While this system investigates different execution paths, virtualization is not one of the changes used to effect a different execution.

The detection of analysis environments is not limited to detecting the virtualization component itself. Holz et al. describe numerous techniques for detecting honeypots [127]. Some techniques are indeed rooted in virtualization detection, but others focus on other environmental components such as the presence of a debugger.

The concept of evasion has also been subject to a considerable amount of research in the context of IDS. Handley et al. observe that a skilled attacker can "evade [network] detection by exploiting ambiguities in the traffic stream" [121]. These ambiguities are classified into three categories: incomplete system implementation, incomplete knowledge of the traffic recipient's system, or incomplete network topology knowledge. Numerous others describe evasion attacks due to inconsistencies between the IDS and the victim computer [74, 93, 114, 162, 182].

The most closely related work to ours offers emulator detection techniques for the PC [183]. Raffetseder et al. detail some "general" detection methods and well as a few that specifically detect the Intel x86 version of the QEMU emulator. At a high level, we explore similar general detection techniques targeting the Android emulator. There is scant similar academic research in evading mobile malware analysis; however there have a been a few industry presentations [169, 178] that look at evasion particular to Google Bouncer. Both of these presentations address some API related detections (which we generalize in Section 5.5.3) as well as detections that specifically target Bouncer, such as source IP addresses associated with Google. Our work here is more general, in that it tries to pinpoint more fundamental limitations in dynamic analysis on mobile devices. A more general industry presentation by Strazzere explores several Android API-based detections as well as a specific QEMU detection and is complimentary to our research [211].

5.5.2 Emulator Detection

Fundamentally, the concept of emulator detection is rooted in the fact that complete system emulation is an arduous task. By discovering or observing differences between virtual and physical execution an attacker can create software virtualization checks that can be used to alter overall program behavior. Such differences may be an artifact of hardware state not correctly implemented in the virtual CPU, hardware or software components that have yet to be virtualized, or observable execution duration.

In this section we detail several virtualization detection techniques and discuss the results of experimental evaluation of these techniques. The techniques require few or no permissions and work on commodity devices in the standard consumer software configuration. As with any consumer Android device, applications are governed by Linux access control and are thus limited to user-mode processor execution (i.e., devices are not rooted).

We evaluate the detection techniques using emulator instances on similar Windows, Mac, and Linux hosts (each with an i7 processor, 8 GB RAM) as well as six physical devices from major U.S. cellular carriers. We divide detection techniques into the following categories: differences in behavior, performance, hardware and software components, and those resulting from system design choices. For ease in comparison, the first three roughly coincide with existing work in PC emulator detection [183].

5.5.3 Differences in behavior

Since virtualization is often defined in terms of execution being indistinguishable from that of a real machine, in some sense, any virtualization detection technique can be perceived as a difference in behavior. However, here we focus on behavioral differences specific to software state and independent of system performance.

Detecting emulation through the Android API The Android API provides an abstract interface for application programmers. Since many Android devices are smartphones, the API provides a rich interface for telephony operations as well as methods for interacting with other hardware and local storage.

Table 5.2 enumerates several API methods that return particular values when used with an emulated device. Each of the API-value pairs in Table 5.2 can be used to explicitly detect an emulated device or used in conjunction with other values in order to determine a likelihood. For example, if the `TelephonyManager.getDeviceId()` API returns all 0's, the instance in question is certainly an emulator because no physical device would yield this value.

API method	Value	meaning
Build.ABI	armeabi	is likely emulator
Build.ABI2	unknown	is likely emulator
Build.BOARD	unknown	is emulator
Build.BRAND	generic	is emulator
Build.DEVICE	generic	is emulator
Build.FINGERPRINT	generic††	is emulator
Build.HARDWARE	goldfish	is emulator
Build.HOST	android-test††	is likely emulator
Build.ID	FRF91	is emulator
Build.MANUFACTURER	unknown	is emulator
Build.MODEL	sdk	is emulator
Build.PRODUCT	sdk	is emulator
Build.RADIO	unknown	is emulator
Build.SERIAL	null	is emulator
Build.TAGS	test-keys	is emulator
Build.USER	android-build	is emulator
TelephonyManager.getDeviceId()	All 0's	is emulator
TelephonyManager.getLine1 Number()	155552155xx†	is emulator
TelephonyManager.getNetworkCountryIso()	us	possibly emulator
TelephonyManager.getNetworkType()	3	possibly emulator (EDGE)
TelephonyManager.getNetworkOperator().substring(0,3)	310	is emulator or a USA device (MCC)‡
TelephonyManager.getNetworkOperator().substring(3)	260	is emulator or a T-Mobile USA device (MNC)
TelephonyManager.getPhoneType()	1	possibly emulator (GSM)
TelephonyManager.getSimCountryIso()	us	possibly emulator
TelephonyManager.getSimSerial Number()	89014103211118510720	is emulator OR a 2.2-based device
TelephonyManager.getSubscriberId()	310260000000000‡‡	is emulator
TelephonyManager.getVoiceMailNumber()	15552175049	is emulator

Table 5.2: **Listing of API methods that can be used for emulator detection.** Some values clearly indicate that an emulator is in use, others can be used to contribute to likelihood or in combination with other values for emulator detection. † xx indicates a small range of ports for a particular emulator instance as obtained by the ADB. Emulator instances begin at 54 and will always be an even number between 54 and 84 (inclusive). ‡ 310 is the MCC code for U.S. but may also be used in Guam. †† The value is a prefix. ‡‡ An emulator will be in the form MCC + MNC + 0's, checking for the 0's is likely sufficient.

Similarly, emulator instances adopt a telephone number based on the ADB port in use by the emulator. When an emulator instance starts, the emulator reserves a pair of TCP ports starting with 5554/5555 (a second instance would acquire 5556/5557) for debugging purposes. The adopted telephone number is based on the reserved ports such that the initial emulator instance adopts precisely 1-555-521-5554. Therefore, if the `TelephonyManager.getLine1-Number()` API indicates that the device phone number is in the form 155552155xx, then the device is certainly an emulator. Such a number would never naturally be used on a device because the 555 area code is reserved for directory assistance [20]. The presence of the 555 area code may also be used in other emulator detections such as the pre-configured number for voicemail.

Other values in Table 5.2 are certainly used by the emulator but may also be used by some real devices. Consider the MCC and MNC values obtained via the `TelephonyManager.getNetworkOperator()` method. The emulator always returns values associated with T-Mobile USA. Since there are certainly real devices that use the same codes, checks based on the MCC and MNC need to be augmented with other data. If another check establishes that

Device	Fingerprint
Emulator	generic/sdk/generic/:1.5/CUPCAKE/150240:eng/test-keys
Emulator	generic/sdk/generic/:1.6/Donut/20842:eng/test-keys
Emulator	generic/sdk/generic/:2.1-update1/ECLAIR/35983:eng/test-keys
Emulator	generic/sdk/generic/:2.2/FRF91/43546:eng/test-keys
Emulator	generic/sdk/generic:2.3.3/GRI34/101070:eng/test-keys
Emulator	generic/sdk/generic:4.1.2/MASTER/495790:eng/test-keys
Emulator	generic/sdk/generic:4.2/JB_MR1/526865:eng/test-keys
Motorola Droid	verizon/voles/sholes/sholes:2.0.1/ESD56/20996:user/release-keys
Motorola Droid	verizon/voles/sholes/sholes:2.2.1/FRG83D/75603:user/release-keys
HTC EVO 4G	sprint/htc_supersonic/supersonic:2.3.3/GRI40/133994.1:user/release-keys
Samsung Charge	verizon/SCH-I510/SCH-I510:2.3.6/GINGERBREAD/EP4:user/release-keys
Samsung Galaxy Nexus	google/mysid/toro:4.1.1/JRO03O/424425:user/release-keys

Table 5.3: **Listing of `Build.FINGERPRINT`s collected from various instances.** Emulator instances clearly include common substrings not found in physical devices.

the device is a Verizon device, but the MNC shows T-Mobile, this may indicate a modified emulator that is returning spoofed values.

Table 5.2 also contains several values from the `android.os.Build` class which contains information about the current software build. Retail devices will have system properties that detail the actual production build. An emulator will have build properties based on the SDK build process used to create the emulator binary, such as the `Build.FINGERPRINT`. The fingerprints listed in Table 5.3 clearly show a pattern followed by the SDK build process. Even though the SDK documentation warns "Do not attempt to parse this value," testing for the presence of "generic," "sdk," or "test-keys" yields perfect results for emulator detection when compared to our sample of physical devices.

Experiments with physical devices led to some counter-intuitive findings for some values. For example, the `Build.ABI` value on the emulator is "armeabi" which is a plausible value for all devices with an ARM processor (nearly all devices). However, the API returned an empty string when used on a Motorola Droid. Similarly, a `Build.HOST` value starting with "android-test" was also found on the Droid. As shown in Table 5.4, the `Build.HOST` value is not as useful for emulator detection as other `Build` values.

Detecting emulated networking The emulated networking environment is often quite different than that found on physical devices. Each emulator instance is isolated from the host PC's network(s) via software. The network address space is always 10.0.2/24. Furthermore, the last octet of the virtual router, host loopback, up to four DNS resolvers, and the emulator's address are always known (1, 2, 3–6, and 15, respectively). Unlike ADB, which reserves adjacent, incrementing TCP ports, the network schema is the same for every emulator instance, even if several instances are simultaneously running on one host.

While it is possible that a network to which a real device is connected may exhibit exactly the same network layout, it is unlikely. Devices configured with cellular data plans will often use carrier DNS resolvers and have

Device	Build.HOST
Emulator	apa27.mtv.corp.google.com
Emulator	android-test-15.mtv.corp.google.com
Emulator	android-test-13.mtv.corp.google.com
Emulator	android-test-25.mtv.corp.google.com
Emulator	android-test-26.mtv.corp.google.com
Emulator	vpbs30.mtv.corp.google.com
Emulator	vpak21.mtv.corp.google.com
Motorola Droid	android-test-10.mtv.corp.google.com
HTC EVO 4G	AA137
Samsung Charge	SEI-26
Samsung Galaxy Tab7	SEP-40
Samsung Galaxy Nexus	vpak26.mtv.corp.google.com

Table 5.4: **Build values collected from various instances.** For several versions of the Android emulator and several physical instances, patterns in `Build.HOST` are easy to identify.

a carrier Dynamic Host Configuration Protocol (DHCP) lease for the device IP. WIFI networks to which a device connects are also unlikely to be configured in exactly this way, and are unlikely to exhibit the quiet nature of a /24 network solely allocated to one emulated device.

Accessing network information is relatively straightforward using standard Java techniques. A trivial detection for the emulated network would be to check for the four known IP addresses. The false positives on this approach would be low and suffice in most situations. In order to obtain networking information and to interact with the network, the Android application would need to request the `ACCESS_NETWORK_STATE` and `INTERNET` permissions.

Detecting the underlying emulator The underlying emulator, QEMU, is employed to drive the hardware emulation of the Android emulator. As such, any QEMU detection techniques such as using CPU bugs [183] or virtual address allocation [104] can also be employed against the Android emulator. These techniques, however, require to run so-called native code, that is, software that executes on the processor instead of Android's Dalvik Virtual Machine (VM). Such software must be compiled with the Native Development Kit (NDK), and until recently was on found in 4.52% of applications [249]. Hence, malware that attempts to detect the underlying emulator using native code may actually be easy to flag as suspicious. Now, as much as 70% may employ at least some native code [232], so there may increasingly be opportunity for these types of evasions.

5.5.4 Differences in performance

The emulator, unassisted by hardware, is at a natural disadvantage when it comes to processing speed. Translating the instructions for execution indeed inherently causes a performance penalty. However, the emulator is typically run on a PC with considerably more processing resources than a mobile device. Thus, it is plausible that, even with the instruction translation penalty, an emulator could approximate the performance of a physical device. We tested

Device	Average Round Duration (Seconds)	Standard Deviation
PC (Linux)	0.153	0.012
Galaxy Nexus (4.2.2)	16.798	0.419
Samsung Charge (2.3.6)	22.647	0.398
Motorola Droid (2.2)	24.420	0.413
Emulator 2.2	62.184	7.549
Emulator 4.2.2	68.872	0.804

Table 5.5: **Pi calculation round duration on tested devices using AGM technique (16 rounds).** The tested devices are noticeably slower at performing the calculations than related devices running similar software.

this hypothesis by (1) measuring CPU performance and (2) measuring graphical (Graphics Processing Unit (GPU)) performance.

CPU performance

Some hardware features often used for performance measurements are unavailable to the application executing in user-mode. For example, the performance counters on ARM processors are enabled via the CP15 register and may only be enabled from the privileged kernel—which will not be accessible unless the phone is rooted. This limitation also poses a substantial barrier to using many processor benchmarking software suites, which require higher privilege than an Android application possesses.

We instead turn to a crude method of benchmarking: evaluating the duration of lengthy computations. We created a Java Native Interface (JNI) application for Android using the NDK. Our application calculates the first 1,048,576 digits of Pi using the method described by Ooura [172]. Pi is calculated over 16 rounds of increasing precision using AGM making extensive use of Fast Fourier Transforms (FFTs), square roots, multiplication, etc.

The AGM calculation shows significantly different results when comparing emulated instances with similar physical devices. Table 5.5 shows the average and standard deviation for several tested devices. For instance, executing the native application on a 4.2.2 emulator results in an median round duration of 68.8 seconds with a total execution time of 1101.9 seconds. A 4.2.2 Galaxy Nexus demonstrates a 16.8 second median round duration, and takes 268.8 seconds to complete. Comparatively, the PC hosting the emulator executes the same code 2.5 seconds with a median of 0.15 second/round (Linux PC). Round durations show statistically significant differences between emulated and non-emulated environments (Wilcoxon test, $W = 256$, $p < .001$), which in turn demonstrates the viability of the this emulation detection technique.

Graphical performance

We also investigated differences in the video frame rate across various hardware and emulator instances. To that effect, we created an Android application that uses Android's `SurfaceHolder` callback method as part of the

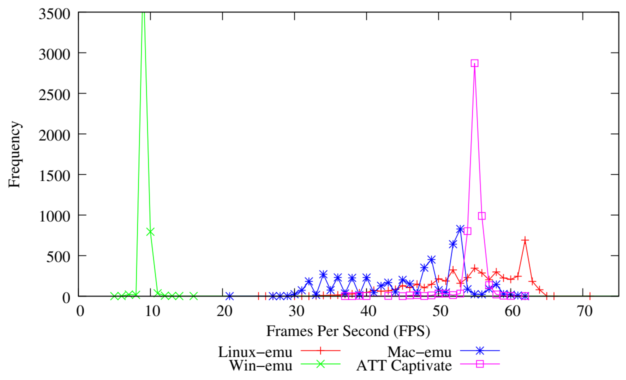


Figure 5.10: **Android 2.2 FPS measurements**: Windows exhibits a very low frame rate. The hardware device, ATT captivate, clearly has a very tight bound on framerate at 58-59 FPS. All emulators are statistically different from the real device (Wilcoxon, $p < .001$). Figure from [226].

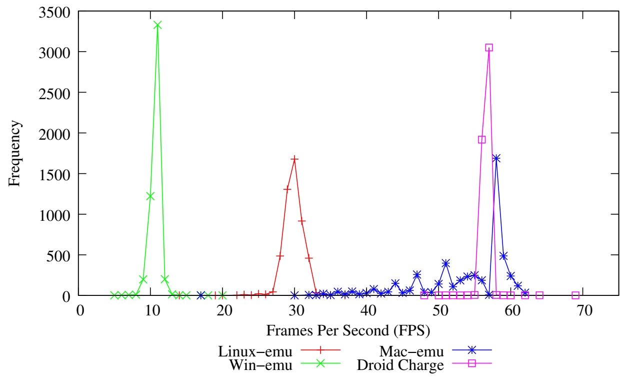


Figure 5.11: **Android 2.3.x FPS measurements**: The Droid Charge registers the majority frame readings between 57 and 58, while the emulators show lower, more distributed readings. The emulators are 2.3.3; the Droid Charge is 2.3.6. All emulators are statistically different from the real device (Wilcoxon, $p < .001$). Figure from [226].

`SurfaceView` class. A thread continuously monitors the `SurfaceHolder` canvas and calculates an Frames Per Second (FPS) value. We sampled 5,000 values for each device.

We ran each emulator instance serially on a 4-core i7 host with 8 GB of RAM and at no time did the the system monitor indicate that any PC resources were a limiting factor. Each emulator and physical device was cold booted and left idle for five minutes prior to testing, and the instances were not otherwise used.

Figures 5.10–5.12 show that physical devices typically exhibit both a higher and much more consistent FPS rate. For example, the Galaxy Nexus (4.2.2) shows 58 or 59 FPS for quite near the duration of the experiment. However, a similar emulator shows both much lower FPS near 11 and exhibits more of a bell curve with observable measurements between 9 and 14. Also, some frame rates simply would not occur on a commodity device, such as the 150+ FPS reported occasionally by a 3.2 emulator.

From the experimental data, heuristics can be devised to detect emulators in a host-OS agnostic way across all versions of Android. One such heuristic is shown in Figure 5.13. The intuition behind the heuristic is that the

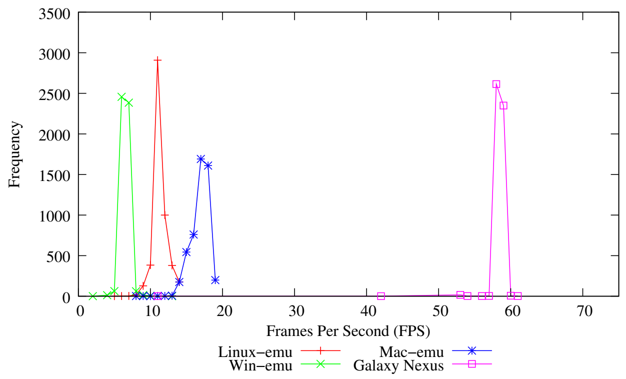


Figure 5.12: **Android 4.2.2 FPS measurements**: Emulators clearly show a low rate, and more of a bell curve than the Galaxy Nexus which shows almost entirely 59-60 FPS. All emulators are statistically different from the real device (Wilcoxon, $p < .001$). Figure from [226].

```
for a value: v
 IF
  > 80% of samples fall within v-1..v+1
 AND
  v is > 30 FPS
 THEN
  the sampled device is a physical device
 ELSE
  the sampled device is an emulator
```

Figure 5.13: **Sample FPS heuristic** using FPS to determine if an Android application is executing on a physical device, indicated be high and closely-coupled FPS values.

emulators generally exhibit slower and less tightly bounded FPS rates, often representing a bell curve. Each of the physical devices, on the other hand, exhibit relatively high and tightly coupled rates. By identifying FPS distributions that are less coupled or slower, emulator instances can be identified.

Versions of the Android SDK after API 17 allow for limited emulator GPU acceleration, though only on Mac and Windows systems. This feature is only meant to work for emulator instances of Android 4.0.3 or later [31], but we tested several configurations anyway. Figures 5.14 and 5.15 show FPS results from Mac and Windows emulators and GPU-assisted emulators.

On the Mac, the 4.2.2 emulator instance, the only supported platform, appears to behave considerably more like a physical device than the 4.2.2 emulator without assistance from the host GPU. However, the GPU-assisted 4.2.2 emulator still registers visible FPS rates in the 30-60 and 60-65 ranges, not the tightly coupled plot of almost exclusively 59-60 FPS as observed in Figure 5.12. The difference between the GPU-assisted 4.2.2 emulator on the Mac and a real Galaxy Nexus, like all of the timing results, is statistically significant (Wilcoxon test, $W = 8.1 \times 10^6$, $p < .001$). On the other hand, GPU assistance on Windows emulators does not considerably improve upon collected values.

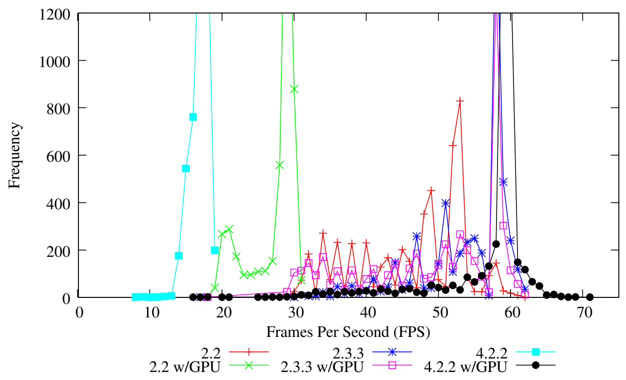


Figure 5.14: **GPU-accelerated emulator (Mac)**: The only supported version, 4.2.2, shows significant improvement when using the host GPU (AMD Radeon 6750M, 1GB RAM). The GPU-assisted 4.2.2 emulator more closely resembles a physical device, but still exhibits a visible tail through the 30-60 FPS range. Figure from [226].

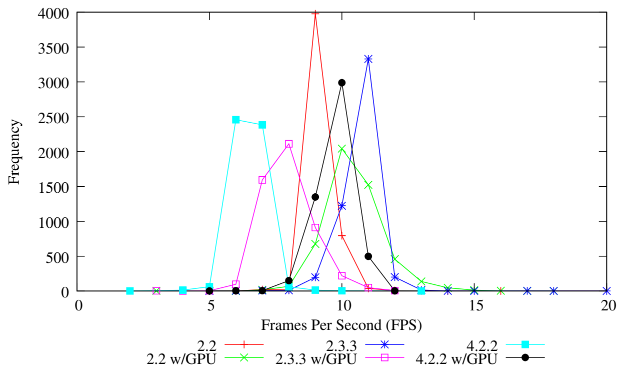


Figure 5.15: **GPU-accelerated emulator (Windows)**: Using the GPU (NVidia 545 GT, 1GB RAM) in Windows did not have a significant effect Note: the scale of this graph is much different than other FPS graphs in this section. Figure from [226].

5.5.5 Differences in components

Modern devices are composed of complex combinations of hardware subsystems that are all meant to work in concert. These subsystems all have their own physical implementations and often vary from what is provided with the Android emulator. Likewise, devices are shipped with proprietary software that drives special hardware, interacts with web services, or implements specific functions such as Digital Rights Management (DRM). These hardware and software components can both be used to differentiate physical devices from virtual instances.

Hardware components

With errors in hardware design, such as CPU bugs, indistinguishable emulation of a processor is an arduous task. Emulation of a complete hardware system is even more difficult. Since emulated environments must appear similar to a physical device, other components such as I/O ports, memory management chips and networking devices must all somehow be made available to emulated software. Similar to virtual IDE/SCSI devices exhibiting certain characteris-

Sensor	Android Version	Moto. Droid	Samsung Charge	HTC EVO 4G	Galaxy Nexus
Accelerometer	1.5	1	1	1	1
Ambient Temperature	4.0	-	-	-	0
Gravity	2.3	-	1	1	2
Gyroscope	1.5	0	1	0	2
Light	1.5	1	1	1	1
Linear Acceleration	2.3	-	1	1	2
Magnetic Field	1.5	1	1	1	1
Orientation	1.5	1	1	1	2
Pressure	1.5	0	0	0	1
Proximity	1.5	1	1	1	1
Relative Humidity	4.0	-	-	-	0
Rotation Vector	2.3	-	1	1	2
Temperature	1.5	1	0	0	0
Total		6	9	8	15

Table 5.6: **Android `Sensor` types,** the earliest version of Android to support each type, and observed sensor counts on four physical devices.

tics that facilitate PC emulator detection [183], differences can be observed in the Android emulator. We focus on two classes of differences, those observable due to emulated hardware (or lack of) and those observable due to omitted software.

Physical components Much like specific hardware values present in PC components, values for various hardware components are observable in Android. For example, the CPU serial number is world-readable as part of `/proc/cpuinfo`. Emulator CPUs always show a value of sixteen 0's, while real ARM CPUs return a unique 16-character hexadecimal string. Similarly, current CPU frequencies can be retrieved from `/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_min_f req` and `max_freq` on a Galaxy Nexus, but these files are not present in a 4.2.2 emulator.

In addition to board-level design decisions such as the use of different memory types [231], devices employ a myriad of motion, environmental and positional hardware sensors often not found on PCs. Even budget devices often have GPS receivers, Bluetooth, accelerometers, temperature sensors, ambient light sensors, gravity sensors, etc. More recent or expensive devices often have additional capabilities perceived as market-differentiators such as NFC chips, air pressure sensors, or humidity sensors.

The sensor types supported as of Android 4.x (API 14) are shown in Table 5.6. Some types of sensors are not natively supported on older devices. Observing the type and quantity of sensors on a particular device can easily be performed via an the `SensorManager` API. The size of the list returned from `getSensorList()` for each type of sensor shown in Table 5.6 reveals the quantity of each type. Even early devices such as the Motorola Droid have many types of sensors.

Simply observing the number of devices may be sufficient for emulator detection, but this metric is relatively easy

```
hw.sensors.temperature=yes
hw.camera.back=emulated
hw.gpu.enabled=yes
hw.gsmModem=yes
hw.sensors.magnetic_field=yes
hw.accelerometer=yes
hw.audioOutput=yes
hw.battery=yes
hw.sensors.proximity=yes
hw.lcd.backlight=yes
hw.sensors.orientation=yes
hw.camera.front=emulated
hw.gps=yes
```

Figure 5.16: **Android Virtual Device (AVD) configuration settings to add emulated hardware.** These settings can be added to the device ini file in order to indicate the virtual presence of hardware sensors such as the accelerometer, or temperature sensor.

to spoof by modifying the SDK. A modified emulator may simply return lists of an appropriate size for each sensor type. More advanced emulator detection could be performed by interacting with each sensor. This type of emulator detection would require considerable modification to the emulator, such as emulating the entire functionality of a sensor along with synthetic data.

Recent versions of the SDK facilitate adding some virtual hardware to an emulator instance. Figure 5.16 enumerates the configuration settings for several supported device types, one of each may be added to an emulator instance. However, this support is limited to later versions of Android, and the degree to which the virtual device emulates a physical device varies.

As an example, we implemented an application to continuously monitor the accelerometer, a very common sensor that requires no permission to be accessed. Since we are interested in gathering data as close as possible to the real-time readings we poll using the fastest setting (`SENSOR_DELAY_FASTEST`). Since the accelerometer measures acceleration (m/s^2) along all three axes, we subtract the acceleration force of gravity (9.81) from the vertical axis to normalize the recorded values. In this way, an accelerometer of a stationary device should register 0 for all three axes.

Figure 5.17 shows measurements for a physical device are closely coupled, but are neither always constant for any axis nor 0. A virtual device with an emulated accelerometer yields exactly the same value on every axis for every sampled data point. Regardless of Android version in the emulator or the host OS on which the emulator is used, the values are always 0.0, 9.77622, 0.813417 (x, y, and z). A device that is actually being used will show a drastically wider distribution along all axes.

Similar detections can be created for other sensors, either to detect an emulator exactly, via known emulator values, or by determining a heuristic for detecting a physical device based on hardware ranges. Furthermore, similar detections can be created for other hardware subsystems such as Bluetooth which is often found on physical devices but is not

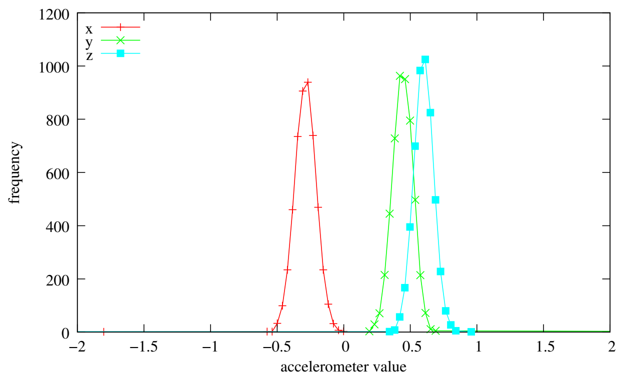


Figure 5.17: **Accelerometer measurements**: Measurements from 5000 data points gathered as quickly as possible from a Samsung Galaxy Nexus (4.2.2) (vertical value adjusted for gravity). Figure from [226].

```
int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
float batteryPct = level / (float)scale;

boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;
```

Figure 5.18: **Battery level emulator detection example.** The battery level is obtained via two Android Intents [18]. If batteryPct is exactly 50% or the level is exactly 0 and the scale is exactly 100, the device in question is likely an emulator. The level could be monitored over time to ensure it varies, and the charging status could be used to determine if the battery should be constant (at 100%).

present at all in the emulator. Simply testing to see if a valid handle is returned from `BluetoothAdapter.getDefaultAdapter()` will indicate that the device has a Bluetooth capability and is thus a physical device.

Software components relating to hardware Detection techniques very similar to the sensor detections discussed above can be created for the camera(s) and for readings akin to sensors such as the battery level. The battery could be monitored over time to ensure the battery level changes or depletes. The exception, of course, is if the device is plugged in and is thus constantly at 100%. The level and charging status can be observed using the `BatteryManager` API as shown in Figure 5.18. The battery level on the emulator is exclusively set at 50% or the two components are two known constants (level is 0 and scale is 100).

Another detection relates to special software added by manufacturers in order to support certain hardware. Manufacturers often add special hardware to a device as a market-differentiator to consumers or in order to take advantage of hardware not natively supported by Android. Such support is added via kernel modules, and like many Linux-based systems, Android detections could consist of looking at what might be loaded from `/sys/module` or the `/lib/modules` directory and what is currently inserted via the `lsmod` command or `/proc/modules` interface.

As with other detection techniques, kernel module detection can take a high-level approach by counting (a 4.2.2 emulator shows 26, a physical Galaxy Nexus shows 72), or a more granular approach.

One specific example of such software is kernel modules added by Samsung in order to take advantage of Samsung memory and the proprietary Robust File System (RFS) [231] instead of the common Linux Memory Technology Device (MTD) system. Simply listing `/proc/modules` on the Samsung Charge reveals that modules for RFS are loaded on the device. To improve detection, the compiled modules (`/lib/modules/rfs_fat.ko` and `rfs_glue.ko`) can be inspected for `modinfo`-like information or for specifically known byte sequences.

Software components

In addition to the software that specifically supports hardware, consumer Android devices are shipped with a variety of additional software. This software ranges from pre-installed applications to services designed specifically to interface with "cloud" resources or enable DRM. Clear omissions from the emulator are the applications and supporting software for Google's Internet services. The marketplace application, Google Play, the Google Maps API, Google Talk, the Google services used to handle authentication and session information, and other similar software are found on nearly every consumer Android device, but are not included in the emulator. For this reason, observing `GoogleLoginService.apk`, `GoogleServicesFramework.apk`, `Phonesky.apk` or `Vending.apk` in `/system/app` likely indicates a real device. Carrier-added applications such as Verizon's backup software (`VZW-BackupAssistant.apk`) can be observed the same way and similarly indicate real device.

Instead of or in addition to inspecting files, an API can be used to query the installed applications in an instance. The `PackageManager`'s `getInstalledPackages(0)` interface can be used to obtain a list of all installed applications. The list can then be queried for the Java-style package names such as `com.google.android.gsf` or `com.android.vending`. The associated application for each list item can also be located indirectly through `applicationInfo.sourceDir` which will provide the full path to the APK.

Android's `ContentResolver` can also be used to query the system for the presence of a software service. For instance, the Google Services Framework identifier can be queried with `ContentResolver.query(content: //com.google.android.gsf.services, null,null,"android_id",null)`. The emulator does not support this service and will always return `null` for this query.

In addition to observing the presence (or absence) of software services, variations in software component behavior can be observed. For instance, when establishing an interactive shell to an instance via ADB, the behavior is different between an emulator and a physical device. In particular, in an emulator shell the effective Linux UID is 0 (root). This difference is caused by a check in ADB called `should_drop_privileges` which inspects two "read-only" system properties: `ro.kernel.qemu` and `ro.secure`. The inspection verifies that the instance is running the emulated kernel and that the "secure" setting is zero meaning that the root shell should be permitted.[3]

There are no APIs for inspecting properties such as `ro.secure`, but the properties are loaded from the `default.prop` file at system boot. Standard Java file methods can be used to read this file and therefor examine settings such as `ro.secure`. Even though this file is not likely to be modified, obtaining the properties in this way may not reflect the runtime state of the instance. We will explore other ways of obtaining actual runtime values in section 5.5.7.

5.5.6 Differences due to system design

Modern runtime analysis systems must cope with certain constraints that do not affect consumer devices. In particular runtime systems must generally process a considerable volume of malware as the number of daily unique samples is already quite large and continues to grow. This phenomenon has been observed for years in the realm of PC malware and early signs indicate a similar growth pattern for mobile malware (see section 3.3). Unfortunately, this requirement for additional scale is often at odds with resource availability. It is simply not economically viable to purchase thousands of physical devices or to run thousands of emulators simultaneously, forever. For these reasons, critical design decisions must be made when designing a runtime analysis system and the effects of these decisions can be used as a form of runtime-analysis detection. We outline two classes of design decisions that may influence an attackers ability to detect or circumvent analysis: those shared with PC runtime analysis systems and those specific to Android.

PC system design decisions, such execution time allotted to each sample or how much storage to allocate to each instance, have been explored in the PC realm. Many of these same decisions must be made for a mobile malware runtime analysis system. System circumvention, such as delaying execution past the maximum allotted execution time, is also shared with PC techniques.

Android-specific design decisions revolve around the inherent differences between a device that is actively used by an individual and a newly created instance (virtual or physical). If an attacker can determine that a device is not actually in use, the attacker may conclude that there is no valuable information to steal or that the device is part of an analysis system.

Metrics for determining if the device is (or has been) in use include quantities such as the number of contacts and installed applications, and usage indicators such as the presence and length of text messaging and call logs. These indicators (and many more) are available programmatically as part of the Android API, but many require the application to request particular application permissions. Runtime analysis system detection using these metrics is

[3]The method also employs a second check for `ro.debuggable` and `service.adb.root`; if these are both 1, privileges will not be dropped even if `ro.secure` is set to 1.

not as clear-cut as the other techniques we presented. These values depend upon knowing the average or minimum quantities present on the typical consumer device, and would be rife with false positives if the quantities were not evenly distributed among all users. Some work shows that these values are indeed not evenly distributed as a small user study showed 40% (8 of 20) of participants downloaded ten or fewer applications while 10% (2 of 20) downloaded more than 100 [134].

5.5.7 Minimizing the permissions needed

Some of the detections mentioned in this paper require certain application-level permissions. For example, to detect if Bluetooth is present on a device the application must request the `android.permission.BLUETOOTH` or `BLUETOOTH_ADMIN` permission. Other resources, such as the accelerometer require no permission to access.

All the techniques described in previous sections require only a very limited set of application permissions and they make use of existing APIs that are unlikely to change substantially in the foreseeable future. However, using the advertised APIs also has a offensive drawback: Designers of automated analysis systems could attempt to address the specific APIs we have utilized instead of completely addressing the downfalls of the emulation environment. For example, mitigating the device Identifier (ID) check by simply hooking the call to `TelephonyManager.getDeviceId()` and causing the return value to not be all 0's. To demonstrate how such a defensive approach is ultimately shortsighted, we present two techniques for retrieving runtime system properties even though there is no programmatic API in the SDK. We can then use these properties to create alternate implementations of nearly every detection we have detailed.

We offer two additional techniques for obtaining system properties: reflection and runtime `exec` (subprocess). Example code can be found in the Appendix 8.2. In the reflection example, the `android.os.SystemProperties` class is obtained via the `ClassLoader`. The `get` method can then be obtained from the class and used to retrieve specific properties. This example retrieves settings we've already discussed, such as the `ro.secure`, the battery level and the `Build` tags. The `exec` example is more general and simply prints a list of all runtime properties being executed by the `getprop` binary and parses the output.

5.5.8 Evaluation

Our techniques were developed in a relatively isolated test environment. We thus need to measure the effectiveness of our techniques against real analysis systems. We identified publicly available Android sandbox systems via literature review, industry referral, and via Internet searches. Candidate systems had to have a public, automated interface to be considered. Ideally, a candidate system also provides a publicly accessible output—typically as some form of report.

Our candidate sandboxes were Andrubis [4], SandDroid [26], Foresafe [12], Copperdroid [6], AMAT [1], mobile-sandbox [17] and Bouncer [145]. Each sandbox presents a slightly different interface, but are all meant to be accessible as web services. Likewise, each sandbox is the result of different levels of developmental effort and therefore embodies various levels of product maturity.

Mobile-sandbox constantly displayed a message indicating that 0 static and 308,260 dynamic jobs were yet to be processed. We were only ever able to observe static analysis output of mobile-sandbox. Similarly, SandDroid seemed to not route or otherwise filtered outbound network traffic, and the SandDroid reports only displayed results of static analysis so it was not possible to test our evasion techniques on SandDroid.

AMAT's online instance does not appear to be in working order, immediately stating that any uploaded application was "not malware." AMAT did not provide any further reasoning as to why this message was displayed, but uploading APK files did result in the overall analysis number incrementing with each submission. When using Foresafe, an action would occasionally fail, and a server-side error would be displayed to the user. Even so, refreshing the page and repeating the action always seemed to solicit the desired effect.

Google's Bouncer does not immediately meet our minimal requirement of having a public interface, but we attempted to include it given its importance on deployed applications. Not much about the inner workings of Bouncer has been made available. Without the ability to directly submit applications for analysis, and without the ability to directly view reports, interaction with Bouncer is widely unpredictable. Indeed, even after submitting several, increasingly offensive, applications into the Google Play marketplace, we never observed any connections coming from Bouncer. It is possible that Bouncer has altered the decision process for selecting which applications to run (e.g., only those with more than 10K downloads, or those with negative reviews) or has been changed to disallow connections to the Internet following other work on profiling the inner workings of Bouncer [169, 178].

5.5.9 Behavior evaluation

As shown in Table 5.7, the SDK and `TelephonyManager` detection methods prove successful against all measured sandboxes. Many of the simple detection heuristics outlined in Table 5.2 are similarly successful. However, some of the `Build` parameters, such as `HOST`, `ID`, and `manufacturer` require a more complex heuristic in order to be useful. Detecting the emulated networking environment was also very successful as the sandboxes all employed the default network configuration.

5.5.10 Performance evaluation

The graphical performance measurements further indicate that all of the measured sandboxes are built using virtualization. Figure 5.19 shows measurements from the sandboxes as well as a hardware device. As with the emulators sampled in section 5.5.4, each of the sandboxes exhibit a lower, loosely coupled values. Unlike in our own test envi-

Detection method	Andrubis	CopperDroid	ForeSafe
getDeviceId()	Y†	Y	Y
getSimSerial Number()	Y	Y	Y
getLine1 Number()	Y	Y‡	Y
MCC	Y	Y	Y
MNC	Y	Y	Y
FINGERPRINT	Y	Y	Y
BOARD	Y	Y	Y
BRAND	Y	Y	Y
DEVICE	Y	Y	Y
HOST	N	N	N
ID	N	N	N
manufacturer	N	N	N
MODEL	N	N	Y
PRODUCT	N	N	Y
serial	Y	N	N
TAGS	Y	Y	Y
radio	N	N	N
USER	N	N	N
NetCountry	y	N	N
NetType	y	N	N
PhoneType	y	N	N
SimCountry	y	N	N
VMNum	Y	Y	Y
SubscriberID	Y†	Y	Y
Networking	Y	Y	Y
bluetooth	Y	Y	Y
ro.secure	Y	Y	Y
sensors	Y	Y	Y
contacts	Y	Y	Y
call log	Y	Y	Y
performance	Y	Y	Y
Total detection methods	24	14	21

Table 5.7: **Evaluation of evasion detections.** An uppercase Y indicates that the system was detected as an emulator, a lowercase y indicates that the system may be an emulator, and an uppercase N indicates that the detection mechanism did not succeed. † This detection was actually designed to detect a particular tool in the same manner as described in section 5.5.3, and we discuss the detection in the section 5.5.13.‡ the number is not exactly an emulator number, but the area code is 555 which is not valid.

ronment, we have no control over the duration of execution in the measured sandboxes. Due to premature termination, we only received a subset of the 5,000 measurements the application should have generated (604, 814, and 229 for Andrubis, CopperDroid, and Foresafe, respectively).

5.5.11 Hardware and software component evaluation

None of the measured sandbox systems made an effort to emulate hardware sensors. As shown in Table 5.8, only CopperDroid handles accelerometer events, and no sandbox handles any other sensor type. Likewise, the sandboxes report exact, unchanging values for the battery level (0) and scale (100) as used in the emulator.

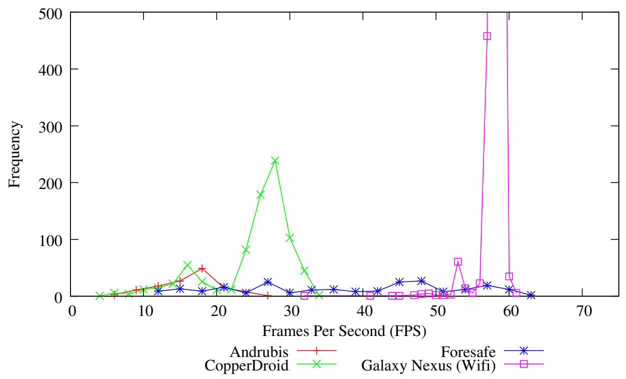


Figure 5.19: **FPS measurements for sandboxes**: For comparison, a physical Galaxy Nexus was re-measured using the same application. The physical device shows strong coupling at 59 FPS and all of the sandboxes demonstrate loose coupling and wide distribution, indicating that they all make use of virtualization. Figure from [226].

Sensor	**Andrubis**	**CopperDroid**	**ForeSafe**
Accelerometer	0	1	0
Ambient Temperature	0	0	0
Gravity	0	0	0
Gyroscope	0	0	0
Light	0	0	0
Linear Acceleration	0	0	0
Magnetic Field	0	0	0
Orientation	0	0	0
Pressure	0	0	0
Proximity	0	0	0
Relative Humidity	0	0	0
Rotation Vector	0	0	0
Temperature	0	0	0
Total	0	1	0

Table 5.8: **Sandbox `Sensor` counts,** as evaluated from different Android sandbox systems. Very different from table 5.6, the sandboxes exhibit very few sensors.

5.5.12 System design evaluation

We elected to only measure coarse data in regard to potentially sensitive items such as address book contacts and call logs. With the wide distribution of our test applications to public sandboxes and Google Play, we wanted to reduce the risk of incidentally receiving any private information. For this reason, we only collected the total counts of contacts, call logs, and applications installed. Even so, these total counts are likely enough to discern if a device is realistically in use as the sandboxes report very low values. Better heuristics could certainly be developed over time. Andrubis reports 63 applications installed and Foresafe reports 46 apps, but neither sandbox has any of the Google services apps such as Google Talk, YouTube, or Google Service Framework installed. Andrubis shows five contacts and three call logs, indicating that some thought has been given to making Andrubis look like a device in use. The standard emulator has no contacts and no call logs, which is also what Foresafe shows.

In addition to the duration shown in the various sandbox reports, we attempted to measure the execution time. We did this by creating an application that embedded the system timestamp (`System.currentTimeMillis()`) in repeated User Datagram Protocol (UDP) packets sent out from the sandbox. By comparing the largest and smallest value received, we can approximate the duration of execution. For instance, Foresafe may report a "Processing Time" of 106 seconds, yet for the same submission we observe timestamp values indicating an execution time of 68 seconds. Similarly, when Andrubis reports an analysis duration of 306 seconds we observe 248 seconds and when CopperDroid reports 731 seconds, we observe 399 seconds. The additional time listed by each sandbox may include some operations other than exclusively executing the application (i.e., sandbox "overhead").

5.5.13 Evasions Discussion

Some of the emulation detections presented in this section are relatively simple to mitigate. For instance, unexpected values could be returned for many of the API methods listed in Table 5.2. Some security tools, such as DroidBox [138], have started to take this step. However, DroidBox does not raise the bar very high as the new values are set to a known constant. For example, the `TelephonyManager` device identifier is always 357242043237517, and the subscriber identifier is always 310005123456789 [7]. When other security tools make use of software packages such as DroidBox, they become subject to evasion due to DroidBox detections. Such is the case for Andrubis. A sandbox system designer must be thoughtful about *how* these detections are mitigated. As demonstrated in section 5.5.7, reflection and runtime exec can be used to observe many system values in different ways, a good mitigation will handle all such cases.

Perhaps the easiest mitigations to counter are those that rely on the high-level state of the device, such as the number of contacts and the call log. Populating a sandbox with a copious address book makes such a detection considerably more difficult for an attacker.

Yet other detection mechanisms we have presented are far more difficult to counter. Techniques rooted in detecting virtualization itself, such as those we presented via timing, present a difficult hurdle for mobile sandbox system designers as they essentially require to redesign the emulator to obtain timing measurements that closely resemble those obtained on actual hardware. While it may be possible to reduce the wide discrepancy we have observed through our measurements, one can easily imagine the next step of this arms race would be to build up a hardware profile based on various measurements (CPU, graphics, ...) over several benchmarks rather than the simple Pi calculation we relied upon. We conjecture it could be possible to pinpoint the exact hardware used with such a technique—and of course, to detect any emulation.

5.6 Discussion

We have presented a system, A5, to completely automate the dynamic execution of Android malware. Our system extracts information from the malware prior to execution in order to better understand how to interact with the malware. In this way, A5 is able to better coerce malicious behavior from the malware and ultimately capture network threat indicators. These indicators can be used simply to quickly, better understand the malware, and to inform network-oriented countermeasures.

The implementation we have described uses novel methods of interacting with mobile applications, extracting interaction points during static analysis to inform the dynamic analysis process. The distributed, parallel design of A5 allows instances to scale with the growth of mobile malware. A5 is not only highly parallel but also modular in design, allowing wholesale replacement of components in favor of newer, better performing options.

As with many malware-related technologies, the detection of dynamic analysis systems is one side of an arms race. The primary reason emulator detection is more applicable here than for PCs is that practical use cases have developed for virtualizing general purpose computers—a phenomenon that has yet to occur for mobile devices. Virtualization is not broadly available on consumer mobile platforms. For this reason, we believe that mobile-oriented detection techniques will have more longevity than corresponding techniques on the PC.

We have also presented a number of emulator and dynamic analysis detection methods for Android devices. Our detections are rooted in observed differences in hardware, software and device usage. From an implementation perspective, the detection techniques require little or no access beyond what a typical application would normally be granted. Such detections can considerably raise the bar for designers of dynamic analysis systems as they must universally mitigate all detections. Of those, hardware differences appear to be the most vexing to address: a very simple evaluation of the frame-per-second rate immediately led us to identify malware sandboxes, without requiring advanced permissions. Likewise, accelerometer values would yield definitive clues that the malware is running in a sandboxed environment. Whether concealing such hardware properties can be done in practice remains an open problem, on which we hope our work will foster research, lest malware sandboxes be easily detected and evaded.

Any sandbox implementation for Android, including A5 must be aware that current virtualization capabilities are incomplete, and analysis will likely soon encounter malware that employs virtualization detection techniques. A5 aims to partially address this issue both by making virtual instances more evasion-resistant and by having the flexibility to use actual physical devices during dynamic analysis.

We also presented performance measurements of a specific implementation of A5 using a public dataset of 1,260 unique Android malware samples. On modest hardware, the implementation was able to process the malware set averaging 149 seconds per sample.

A5 represents a new generation of automated analysis able to cope with large volumes of mobile malware. A5 is able to better coerce malicious behavior by interacting with mobile malware in mobile-specific ways. The ability for A5 to emit network threat indicators, additionally makes A5 immediately useful to cellular providers and operators of wireless networks.

Chapter 6

Secure Software Delivery

Online application stores or "markets" are becoming an increasingly important vector of software distribution (measurements provided in section 3.3). For instance, Apple's flagship MacOS X operating system is, since version 10.7, only distributed through the Apple App Store, thereby entirely forgoing the traditional distribution channel—packaged optical media sold in brick-and-mortar stores. Likewise, the Google Chrome Web Store is a consolidated place to download all extensions to the Chrome web browser.

While their importance is growing, for personal computers application markets are a relatively recent development,[1] and still merely represent one of several alternatives. On the other hand, application markets have been the primary (if not the only, for most users) means of acquiring and installing software on smartphones and tablets.

"Official" application markets for mobile devices, such as Google Play or the Apple App Store act as a centralized software distribution point for a given platform, and allow users to find, download and install applications through a single interface.

Besides official markets, a large number of third-party (or *alternative*) markets exist. Users may rely on these alternative markets, for a variety of reasons, including the unavailability of the official market in a particular country, name-brand recognition (e.g., Amazon's Appstore), or to freely obtain applications that require payment in the official market. Some markets are also locale specific, where existing applications are modified and redistributed for localization purposes. For instance, popular applications may be translated in languages that they do not natively support.

Markets adopt several techniques to provide users with confidence that they are downloading safe applications. First, usually, applications must be cryptographically signed so that their providers are authenticated. Second, markets enforce policies to deal with malicious applications. Some markets (e.g., Apple App Store) vet applications prior to publication [64]. Others, such as Google Play, allow relatively unmoderated publication, but react to identified

[1]The App Store first appeared on MacOS 10.6.6 in Jan 2011.

malware by removing it both from the market and from all (connected) devices that have already installed the malicious application [199].

Unfortunately, these techniques fall short of providing strong security guarantees. When application signatures are certified by the market proprietor (e.g., Amazon and Apple markets), the user has to completely trust the market proprietor to manage and secure the certificates. The fact that existing centralized vetting systems have shown to be imperfect in keeping malware at bay [56–58] seems to indicate that the security guarantees provided by such centralized systems are relatively weak.

In Google Play, the security guarantees are even weaker. Certificates are typically self-signed and, thus, are not bound to any particular identity. Almost anybody can upload applications into the market; and it may take time to realize that some harmful applications have been uploaded. Worse, some of the third-party Android markets may not police malware at all. In fact, it may even be in a market's best interest *not* to do so, as the market operators could enjoy revenue from infected applications.

In other words, existing authentication mechanisms for market applications appear insufficient. For instance, grafting viruses onto pirated software is certainly not a new attack; yet, the lack of proper authentication allows miscreants to use such techniques to distribute malware through application markets.

In this chapter, we propose and evaluate a simple authentication protocol for market applications, that can be immediately deployed on Android, piggy-backs on the naming conventions used for Android packages and applications, and would make it substantially more difficult for an attacker to perform application repackaging.[2]

The remainder of this chapter has the following structure. We first discuss application repackaging techniques in section 6.1. In section 6.2, we provide a novel application authentication mechanism and present a proof-of-concept mobile application, AppIntegrity, which implements such a verification mechanism. We provide a security analysis of our mechanism in section 6.3, and evaluate AppIntegrity through user studies in section 6.4. We discuss related work in section 6.5. We discuss future work and potential AppIntegrity extensions in section 6.6, limitations in section 6.7, and conclude in section 6.8.

6.1 Application Repackaging in Android

We next summarize how application repackaging is performed in Android. To do so, we first describe the structure of an Android application, before turning to repackaging mechanics.

[2] Portions of this chapter previously appeared in [225].

6.1.1 Android applications

In Android, applications are usually written in Java (although some have "native" C components), and are distributed as APK files. Those APK files are in fact Zip archives, which contain compiled Java classes (in DEX format), application resources, and an `AndroidManifest.xml` binary XML file containing application metadata. The APK also contains a public key and its associated X.509 certificate, bundled as a Public Key Crypto-System (PKCS)#7 message in Distinguished Encoding Rules (DER) format.

Naming conventions. When creating a new project, the Android developer documentation dictates that a full "Java-language-style" package name be used, and that developers "should use Internet domain ownership as the basis for package names (in reverse) [45]." This creates package names such as `com.google.maps` for the mobile Google Maps application. To avoid name conflicts, package names must be unique across the entire universe of applications. Using reversed domain names theoretically limits potential namespace conflicts to a developer's own domain.

Signing applications. All Android applications must be cryptographically signed by the developer; an Android device will not install an application that is not signed. Typical Java tools, such as `keytool` and `jarsigner`, may be used to create a unique keypair and sign the mobile application.

In Android, the only key distribution mechanism used consists in bundling developer's public key with the application. Further, Android has no requirement for a keypair to be certified by a Certificate Authority (CA). In fact, we observed that more than 99% of the 76,480 applications detailed in section 3.3 use self-signed certificates.

In other words, the primary purpose of the keypair is to distinguish between application authors, but not to provide any stronger security properties. In practice, keypairs are also used to (1) ensure that applications allowed to automatically update are signed by the same key as the previous version, (2) potentially allow applications signed by the same key to share resources, (3) grant or deny permissions to a family of applications signed by the same key, and (4) to remove all applications signed with the same key from the Android market and potentially from all connected devices when one of these applications is flagged as malware [199].

On the other hand, due to the absence of any CA or PKI, signatures on Android do not provide any assurance about the identity of the signer. Shortly stated, Android ensures that the Facebook application is correctly signed by somebody, but cannot prove the Facebook company actually signed the Facebook application.

6.1.2 Application Repackaging

An existing application redistributed with a different signing key, often with functionality not present in the original version, is said to be *repackaged*. Some, all, or none, of the application's existing functionality can be preserved in the repackaged version.

Applications can be repackaged for many reasons other than to distribute malware. For instance, a repackager may simply wish to add advertising to an existing application to profit from somebody else's application. Application repackaging falls broadly in two classes: spoofing and grafting.

Spoofing. Mobile applications can simply be published under false pretenses, *spoofing* little or none of the features a legitimate application would possess. To deceive the user, a malicious program may advertise to be an existing application, or a nonexistent application that may plausibly exist, yet provide none of the expected functionality. As previously shown in peer-to-peer networks [82] and search-engine result poisoning [132,160], this type of attack could flood a market with enough false positives to attract users.

As an example, in July 2011, the legitimate Netflix application only supported specific devices and versions of Android. Unsupported devices could not locate and install the official application in the market. In October 2011, a fake version of the Netflix application was published in the official Android Market, claimed to "support" all devices, and thus appeared to owners of devices that could not download the legitimate application. The fake application displayed a plausible login screen, but then simply stole service credentials. Once credentials were entered, the application uninstalled itself [59].

Grafting. To achieve the desired functionality of a legitimate application, an attacker may elect to graft malware onto an existing application, and subsequently republish the modified application.

The attacker starts by downloading and extracting an existing application. To do so, she unzips the APK archive to extract the application components (class files and manifest).

Then, she adds malware to the application, and repackages it. Adding malware may require to reverse the DEX-formatted Java classes. While not entirely straightforward, tools such as `undx` [197], `baksmali`, `dedexer`, or `ded` [98] can often successfully decompile `.dex` files to source code. DEX can also be converted to a typical Java jar collection of classes using the `dex2jar` utility, at which point a typical Java decompiler can be used.

In the quite common case in which the `.dex` file does not need to be fully reversed to source code, much of the disassembly and repackaging process can be automated. For instance, `apktool` [43] can unpack and repackage an existing `.apk` with two commands. `apktool` has several side effects that result in non-required changes to the repackaged `.apk`. For instance, some files may be compressed in the repackaged application regardless of whether or not original file was compressed. With automatic compression the repackaged file may actually be smaller than the original despite the addition of malicious code. These side effects may be undesirable for an attacker that wishes for the application to remain as similar as possible to the original application.

In addition to the class files, the attacker may need to modify the `AndroidManifest.xml`, since this is where application-level permissions are specified. This can be done using a tool such as AXMLPrinter2 [46]. For instance, the malware to be added to the existing application may require the `INTERNET` or `SEND_SMS` permissions, even if the permission is not specified in the original application.

Last, prior to publishing the "new" application to one or more markets, the attacker must cryptographically sign the application. The signing can easily be performed with standard Java tools, e.g., using `jarsigner`. Since Android uses self-signed certificates, such signatures will pass installation-time checks.

6.2 Toward Application Verification

Regardless of application vetting policies, it is possible that an application can be repackaged and published into marketplaces that users will frequent. Yet users have no way of knowing if an application claiming a particular origin is in fact created by the assumed author. Here we present a very simple protocol for end-to-end application verification, and discuss an example implementation. The idea behind the protocol is that, while not a panacea against all attacks, it raises the bar that attackers have to clear to be able to carry out spoofing attacks, while being essentially freely deployable with the current Android infrastructure.

In the context of this chapter, verification means that the application is *authenticated*, and that, as a result, its *integrity* against repackaging by third-parties is guaranteed.

6.2.1 Protocol

Prior to publishing, an Android application must be cryptographically signed. This signing makes use of the private key of a keypair generated by the developer. The existence of a keypair provides developers and users with the primitives required to perform other PKCS actions. In particular, the protocol described below takes advantage of the well-known ability to verify a signature. That is for a keypair: secret signing key ssk and public verification key pvk, that the signing of data d results in signed data sd:

$$sd = sign_{ssk}(d) \ .$$

Furthermore, that signed data can be verified using only the associated public key:

$$verify_{pvk}(sd) = true \ .$$

It is assumed that the ssk is selected uniformly at random from the set of all possible keys, and that without the ssk it is computationally infeasible to create an sd' that can be verified.

If the developer makes the pvk widely available, it can be retrieved and used to locally verify signed applications. The public key used for signing an Android application, embedded verification key evk, is included in the APK. In most cases, absent of repackaging, $evk = pvk$. In order to deter developer impersonation in repackaged applications, the *pvk* should *not* solely be published via the marketplace from which the application is obtained. If this were permitted, unscrupulous persons could simply continue to repackage applications and provide new evk' keys along with new applications when published. Instead we propose that the author's *pvk* be stored in a predefined location on the author's web server or use methods similar to Domain Key Identified Mail (DKIM) [140] to provide the *pvk* via DNS (or both). In both cases the verification is tied closely to the DNS controlled by the publisher. Again, to deter repackaging, this DNS location must not be specified in a hidden manifest, but must be closely coupled to information presented to the user. As mentioned in section 6.1, application package namespaces "should use Internet domain ownership as the basis... (in reverse)." Therefore, by reversing the package name, a URL can be constructed to locate the *pvk*.

If developers honor the direction to name packages appropriately (83% of applications in Google Play already conform to this convention), the *pvk* can be unambiguously located relative to the URL corresponding to the package name. Suppose that, by convention, the *pvk* is stored in a file `android.cert` at the root of the domain. For instance, an application with the package name `com.facebook.katana` would be signed using an *ssk* that has an associated *pvk* available at `http://facebook.com/android.cert`. The use of domain ownership for key publishing permits the use of self-signed certificates making this protocol immediately deployable. The use of CA's and other PKI infrastructures remain (at this point) optional.

Propagating key information via DNS has certain performance benefits compared to storing the public verification key on a webserver, which must serve the key to every mobile device at verification time. Both methods are however susceptible to various attacks discussed in section 6.3.

By decoupling cryptographic signing from the distribution mechanism, the application can be verified independent of how an application is obtained. Applications obtained via unmoderated file sharing forums, will still bear a package name plausible to the user (e.g., `com.facebook.katana`), and the device will attempt to verify the application with the legitimate Facebook *pvk*. If the application had been repackaged, the verification will fail. If the verification succeeds, it was signed by the owner of the `facebook.com` domain.

Application verification should take place as part of the installation process. Install-time verification can prevent undesirable applications from ever executing on the mobile device. Figure 6.1 shows a timing diagram of the entire verification process. As previously described, the publisher's keypair is created orthogonal to verification. First a user locates an application in some mobile marketplace and the application is downloaded to the mobile device via whatever mechanism the marketplace supports. Once the entire application is downloaded, the embedded signatures can be checked to be well-formed (locally on the mobile device). Next the package name is extracted from the application and reversed in order to determine the location of the *pvk*. The *pvk* is retrieved from the publisher's server (or from DNS). The application is then verified using the *pvk*. If the verification succeeds the application is installed using the typical platform installation process. If verification fails, the application is not installed. Of course, when

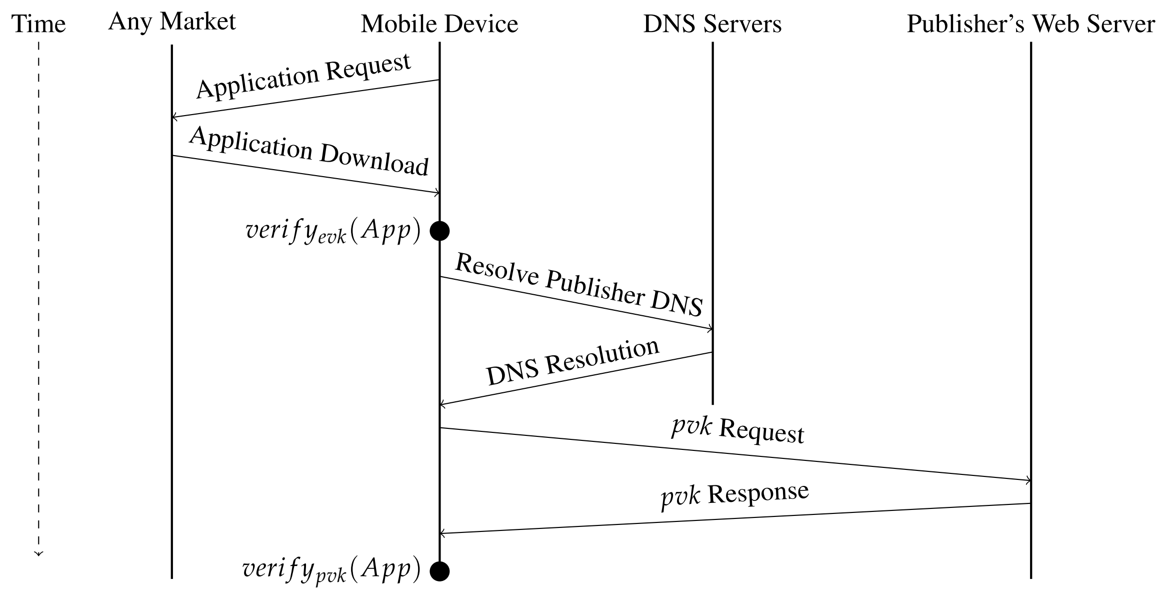


Figure 6.1: **Protocol timing diagram**: Timing diagram for the network actions of the AppIntegrity verification protocol. First an application is requested and downloaded from an online marketplace. Once downloaded, the application signature is cryptographically verified using the embedded key evk. The application package name is then reversed to determine a domain for retrieving the public verification key pvk. The pvk is retrieved from the publisher's web server, or directly via DNS (not depicted), and the application is verified using pvk.

applications fail verification, the implementation could be adapted to permit the user to install the application anyway, or to upload the file to a security team for analysis.

Occasionally, publishers may need to update to a new, legitimate keypair resulting in a new pvk becoming available. Similarly, if a publisher's ssk is compromised a repackaged application may be installed with an pvk that is thought to be valid at install time, but was later found to be compromised. For these reasons, application verification may also be performed periodically or on-demand. Similar to the end result of failed verification at install time, a failed verification in this case would likely result in the uninstallation of an application.

A verification process such as described here is independent to any vetting process imposed by a market policy. The cryptographic verification simply demonstrates that an application is what the publisher intended to provide to the consumer. It makes no attempt to determine if the behavior of an application is malicious.

With this protocol, the end-device is able to verify that the application is exactly what the publisher intended for the user. In the physical world, in addition to trusting the integrity of the store, there is some independent binding to the creator of the software. This binding takes many forms such as product packaging, branding, holographic CDs, and other anti-piracy technologies. The protocol we propose, called AppIntegrity, enables similar binding to take place on modern application markets, creating a way to bind a domain owner to a particular application.

Publishers currently do not make the pvk available independent of an application, as a consequence it is not possible to fully test the mitigation capabilities of AppIntegrity. However, our measurements (by domain validity

Algorithm	Official	Alternative	DKIM compatible
MD5withRSA	9.784%	7.553%	N
SHA1withDSA	2.662%	2.743%	N
SHA1withRSA	87.458%	87.157%	Y
SHA256withRSA	0.091%	2.543%	Y
MD2withRSA	0.005%	0.004%	N

Table 6.1: **Signing algorithms observed in measured markets.** The vast majority of applications utilize DKIM-compatible algorithms already (SHA1withRSA and SHA256withRSA).

checks and manual malware analysis) and the repackaging classification techniques in [248] suggest that AppIntegrity may see great success in mitigating current malware.

6.2.2 Implementation

The protocol described in section 6.2.1, is realized in proof-of-concept applications designed to run on any computer and as an Android application, which, mirroring the name of the proposed protocol, we also dub AppIntegrity. Android's architecture permits the entirety of an application to be observed by other applications. Accordingly, AppIntegrity registers a handler for the `PACKAGE_ADDED` intent that performs verification whenever new applications are downloaded.[3] Since most publishers have not made public keys available, a failed verification results in giving the user the choice to uninstall the application.

AppIntegrity takes advantage of several Android features:

1. Application package names are intended to be unique and based upon domain ownership of the developer.

2. Android applications have read access to other applications. While each application can store data in a private area, the application itself may be read by other applications. Thus a verification program has the ability to obtain package name and signature information from other applications.

3. Android applications are written in Java which has extensive cryptographic libraries that can be used to verify signatures.

4. The Android documentation specifies RSA when generating a private key. The use of RSA in key creation results in a SHA1withRSA (see Figure 6.2) signature, which is compatible with the existing specification for DKIM (DKIM specifies use of RSA-SHA-1 or RSA-SHA-256 for signing and verification). As seen in Table 6.1 the majority of applications we observed (89.7%) use one of the two DKIM compatible algorithms.

[3]Note that this is not exactly the same as the described implementation, since the application is technically already installed by the time the `PACKAGE_ADDED` intent is broadcast. This intent is the nearest to the desired functionality that a typical, unprivileged application can achieve.

```
$ keytool -printcert -file CERT.RSA
Owner: CN=First Last, OU=Unk, O=Unk, L=City, ST=State, C=US
Issuer: CN=First Last, OU=Unk, O=Unk, L=City, ST=State, C=US
Serial number: 4d895f96
Valid from: Tue Mar 22 22:48:54 EDT 2011 until: Sat Aug 07 22:48:54 EDT 2038
Certificate fingerprints:
 MD5:  07:E4:51:41:E8:80:92:97:F9:6F:AF:BF:57:2F:28:2A
 SHA1: D5:A0:3D:A4:E5:0F:D7:9E:B3:53:95:83:8C:CA:AB:A5:EB:E2:C4:29
 Signature algorithm name: SHA1withRSA
 Version: 3
```

Figure 6.2: **Example Android signing key**: Keytool output for signing key. The Signature algorithm name can be clearly identified as SHA1withRSA.

Future implementation could take many forms. To fully realize the protocol shown in Figure 6.1, the Android framework needs minor modification to the install process. Either verification must be built into the package installation process or a new intent needs to be broadcast post-download prior to package install. Meanwhile, the proof-of-concept application may be downloaded from the author's website.[4] Device carriers or manufactures may choose to install a verification application in such a way that the user cannot uninstall it, forcing verification to occur.

6.2.3 Performance Evaluation

Minimal network overhead is crucial as many carriers now have limited data plans. Currently, a typical RSA key found in the official market averages 922 bytes (0.0008 MB). Given the current distribution of application sizes (as shown in Figure 3.3), the additional network overhead introduced by verification is marginal. Figure 6.3 is scaled to show the only appreciable overhead introduced by obtaining the certificate. As seen in the Figure 6.3, less than 4% of applications would exhibit considerable overhead relative to downloading the application. The applications in question are simply so small that the additional 922 bytes is consequential, however it is extremely likely that the user is already downloading many other [134], larger applications further reducing any concerns of network performance degradation.

Similarly, processor use directly affects battery life on mobile devices [64], and as such, excessive resource use could hinder adoption. Since devices already perform cryptographic signature verification the additional verification is not significantly different. Currently a manifest (different that the `AndroidManifest.xml`) is stored in a special `META-INF` directory along with the public keys in the APK. The device currently verifies the signatures stored in the manifest using the embedded public key (*evk*). With our proposed protocol the same key would be obtained dynamically, but the cryptographic operations would remain the same.

To encourage adoption, public keys could still be included in the APK files, and actually both keys (which may

[4]Source code available at `https://github.com/tvidas/appintegrity`.

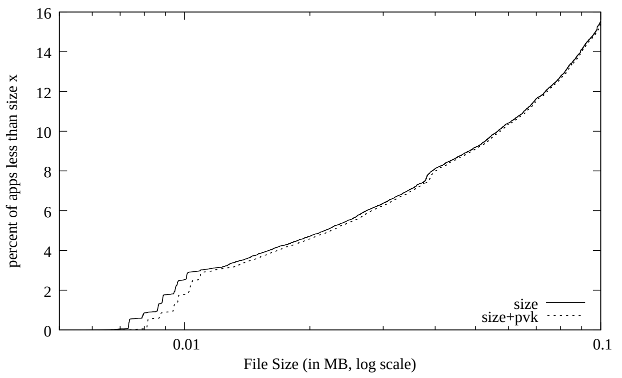


Figure 6.3: **AppIntegrity application size overhead**: Less than 4% of applications would exhibit appreciable overhead (plot magnified from official market plot in Figure 3.3).

be identical) could be used to verify the application. This additional verification would result in a linear increase in processing time as each APK component is verified twice. If the protocol is integrated into the Android package installer, there would effectively be no additional overhead over the existing installer.

The lightweight cryptographic verification of AppIntegrity will likely outperform other types of “fingerprint” or “signature”(not cryptographic signature) based security solutions. In particular, AV, symbolic execution, anomaly detection [80], static analysis [194, 195] would all likely require extensive processor and/or memory requirement which are not desirable on a resource-constrained mobile device.

Since little or no modification is required to to the Android framework, there is negligible network and processing overhead, and there is no additional burden to implement a PKI, AppIntegrity can be deployed to Android with little cost.

6.3 Security Analysis

The primary benefit of AppIntegrity is the ability to verify the integrity of a published application independent of how the application is obtained. Under the current model, an attacker needs only to succeed in getting malware onto a device. Typically this is achieved by publishing a malicious application to a marketplace and allowing users to locate and install the application. AppIntegrity substantially increases the effort required for a successful attack. Under this new model, the attacker must also either obtain the original publisher’s secret signing key, be in control of the publisher’s web server, or commit a Man-in-the-Middle (MitM) attack on the publisher’s DNS records and/or web server. In all cases the attacker must now conduct two successful attacks, and the secondary attack requires more effort than application repackaging.

Man-in-the-middle attacks that target the mobile device may be more difficult to conduct than such an attack on a traditional computer. Modern smartphones and tablets can communicate over several medium. A successful MitM attack on the client will either need to predict the specific media that will be used, or will need to conduct simultaneous MitM attacks on all nearby WIFI, 2G, 3G and 4G networks.

The official market enforces unique package names, which incidentally lightly deters the republication of repackaged applications back to the market. An application with exactly the same package name may not be published. In order to republish in the official market, some existing malware, such as DroidDreamLight2, uses capitalization differently in package names. For example, `com.gb.CompassLeveler` vs `com.gb.compassleveler`. Since DNS does not preserve case, the AppIntegrity verification would resolve to the legitimate key, and fail.

6.3.1 Keeping Private Keys Private

As with most PKCS structures, the cryptographic properties provided by the keypair require the private key to remain secret, known only to the owner. Any other party that knows a user's private key can impersonate that user. For these reasons, users typically create their own cryptographic keypair. Contrary to this convention, Amazon's Appstore supplies an account-unique key to the publisher [42]. In this model, Amazon could impersonate any application publisher, and a security breach of the Amazon market would result in all keypairs being compromised. We encourage developers to exercise the option to request the use of a non-assigned key for application signing.[5] As stated in section 6.2.1, to enable independent verification, public keys should not be solely stored alongside applications in a marketplace.

Similarly, smartphone or tablet users that have rooted their device often install entire new OS images known as "custom Read Only Memorys (ROMs)." These ROMs are created and made available by enterprising developers such as the AOSP. The developers of these ROMs may chose to publicly publish associated private keys. Since the private keys are widely available, no identity can be bound to anything signed by the key. Malware, such as jsmshider [48], may take advantage of this cryptography faux-pas.

Under a model that encourages self-signed certificates, such as the current Android model, the burden of securing the private key falls solely on the application publisher. Publishers that do not properly secure their secret signing key risk others using their identity to publish applications. Under the protocol outlined in section 6.2.1, loss of the private key would allow an attacker to modify an application and have a modified application successfully verify.

6.4 Evaluation

The AppIntegrity protocol aims to provide safety by enabling the end user to verify the integrity of an application irrespective of dissemination model. Some safety can be provided transparently, via technological means, but some parts require user interaction and understanding. Here we briefly cover protocol limitations that are specific to the current

[5]Developers may request to use a non-Amazon key by submitting a request through the Amazon Appstore Developer Portal.

implementation of AppIntegrity, then turn to a deeper discussion of those based on user understanding, bolstered by several user studies in section 6.4.1 through section 6.4.2.

Fundamentally, AppIntegrity would benefit from a few minor design changes to the Android platform: permitting additional privilege to the verification software, enabling actions (in our case, software verification) to perform prior to application install and clearly displaying package name information to users prior to install. The proposed protocol does make several assumptions about users' ability to correctly take action once presented with security-relevant information, and if the user is deceived the effectiveness of AppIntegrity suffers.

Lack of Privilege. As AppIntegrity is currently implemented as a typical Android Application, the application could be uninstalled by a user or potentially by malware. As previously mentioned, a manufacturer or wireless carrier could install AppIntegrity in a way that makes user uninstallation difficult. However, as with most mobile device security properties, rooting the device undermines this added security. Ideally, verification services would be built into Android itself.

Prior Infection. Devices that are already infected with malware that has elevated (root) privileges are subject to other attack classes, such as drawing over the existing user interface. In these situations, AppIntegrity only assists in preventing malware from entering the device. Once malware has infected a device, AppIntegrity is subject to all the same issues that plague infected systems. For instance, malware might disable AV systems, or modify operating system APIs on which AV depends in order to remain hidden.

Domain name deception. Once a user has located a particular application that they are interested in installing, the installation interface must clearly display the package name (or derived domain) to the user. The user may or may not recognize the application name, package name, developer, etc. Even when the user does recognize the application name and developer, it is up to the user to detect *typosquatting*, the intentional registration of domain misspellings [159]. For example, if the user installs a repackaged Google maps look-alike that has a package name `com.g00gle.maps` the certificate will be retrieved from `maps.g00gle.com` and will cryptographically verify. Without an external validation service for URLs (e.g., PKI, reputation system), such attacks will remain possible.

Domain name recognition. Similarly, many users recognize names such as Google, Facebook, etc but the vast majority of applications are created by less recognizable publishers. One may argue that as an application becomes popular, users are more likely to recognize the publisher (and associated domain). However, the problem of unrecognized publishers remains.

AppIntegrity provides a foundation that could be used to create additional protocols or services to help solve this problem. For instance, AppIntegrity would provide assurance that a given publisher produced a certain application, and an external vetting service could assist in confirming that this publisher is reputable. In this way, the relatively unstudied problem of establishing application provenance is transformed into the well-studied problem of domain-based security.

In order to measure the effectiveness of the AppIntegrity protocol, observation of a large implementation would be best. In other words, the best metric to study AppIntegrity would be to implement the protocol with both developers and end users. To encourage this adoption, developers and publishers must be enticed to follow convention, possibly due to requirements imposed by proprietors of large markets.

We evaluated protocol efficacy through user studies. The studies detailed in this section were performed using GCS (see Appendix 8.3 for more detail regarding GCS). GCS was selected over other survey systems primarily due to the native GCS ability to focus on our target demographic—Android users. Not having to manually collect other demographic information explicitly as part of our survey, not having to recruit participants and the speed at which GCS can return results also makes GCS appealing. GCS has been shown to produce highly accurate results, both in comparison to Probability and Non-Probability based Internet surveys [175] and traditional surveys [200]. In most cases GCS outperforms Internet-based surveys (with respect to a benchmark) and is within three percentage points compared to dual-frame telephone surveys.

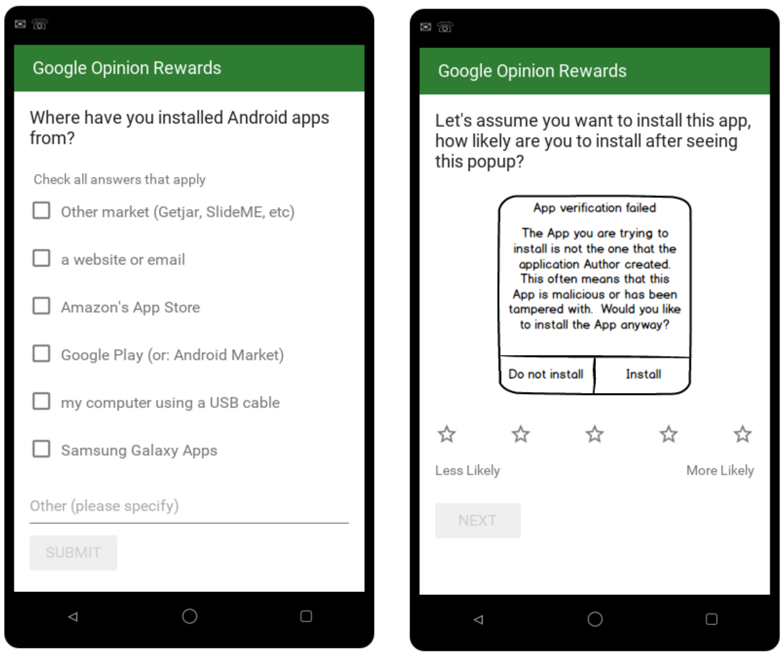
In our studies, participants were expected to take less than one minute to complete the survey, using their Android device. The participant pool was limited to U.S. Android users that explicitly opted-in to the GCS system, and downloaded the related Android App. We further limited this set to those that were 18 or older. Participants were paid between 0 and $1 United States Dollar (USD), variably, as is the case with GCS. Survey question answers were ordered randomly so as to not prime participants.

Our first study (section 6.4.1) addresses several broad aspects of AppIntegrity, confirming the need for such a protocol and examining participant ability to correctly interact with proposed AppIntegrity interfaces. Building upon the first study, we conducted a second study (section 6.4.2) to more deeply address participant domain name deception and recognition abilities.

6.4.1 AppIntegrity Interface Study

Our questions were constructed toward two ends, first, to collect insight about current behaviors regarding device security (rooting) and software market diversity. Besides collecting rooting information as a demographic, it may inform potential AppIntegrity implementations (which may require root access). Surveying market diversity confirms that participants obtain software from multiple sources. Second, our questions were designed to measure participant ability to appropriately take action in reaction proposed AppIntegrity graphical prompts.

From September 7 to 9, 2015, we surveyed seven questions, independently, of 500 participants each:


Google Opinion Rewards

Where have you installed Android apps from?

Check all answers that apply

- ☐ Other market (Getjar, SlideME, etc)
- ☐ a website or email
- ☐ Amazon's App Store
- ☐ Google Play (or: Android Market)
- ☐ my computer using a USB cable
- ☐ Samsung Galaxy Apps

Other (please specify)

SUBMIT

Google Opinion Rewards

Let's assume you want to install this app, how likely are you to install after seeing this popup?

App verification failed

The App you are trying to install is not the one that the application Author created. This often means that this App is malicious or has been tampered with. Would you like to install the App anyway?

Do not install | Install

☆ ☆ ☆ ☆ ☆

Less Likely — More Likely

NEXT

(a) C-MARKET (b) C-FAIL

Figure 6.4: **Google Consumer Surveys: Graphical depiction**: Depiction of the GCS mobile interface. C-MARKET (a) demonstrates a multi-select survey condition and C-FAIL (b) demonstrates a Likert scale condition.

Where have you installed Android Apps from? (Condition *C-MARKET*) In this condition, the participant could select zero or more from six pre-populated and one free-form input answers:

1. Google Play (or: Android Market)
2. Amazon's App Store
3. Samsung Galaxy Apps
4. Other market (Getjar, SlideME, etc)
5. a website or email
6. my computer using a USB cable
7. Other (please specify)

The C-MARKET selections (and a depiction of the GCS interface) are also shown in Figure 6.4(a).

The C-MARKET question was asked to establish that users already currently use markets other than the primary market, Google Play. One can hypothesize that other markets would not exist unless there was some need. For example, there are likely economic incentives for creating and maintaining alternative markets. However, we wanted to perform a measurement to substantiate this hypothesis. We chose the candidate answers via Google search engine rankings from the search "where can I get Android Apps." The top markets were included in the question choices. We included the non-market choices (e.g., USB) to observe the proportion of participants that exhibited these behaviors. We know that some some users must install via USB because malware exists that propagates solely in this way [143]. Similarly, some corporate-controlled Mobile Device Management (MDM) models use non-market installation procedures. For instance, an administrator might sent users an email with a hyperlink to install a corporate mail client.

Have you ever Jailbroken or rooted an Android device? (Condition *C-ROOT*) The participant could select one from "Yes," "No," or "I don't know." This question was asked to gain some understanding as to whether participants familiar with the terms "jailbreak" or "root." If a participant selected yes or no, we assume that the participant believes they understand the meaning of one term. We did not attempt to verify that the participant belief was accurate.

Let's assume you want to install this app, how likely are you to install it after seeing this popup? This question was actually the same for four of the remaining five questions, we only changed the graphic that was presented alongside the question. The next five questions all used a five-point Likert scale labeled "Less Likely" to "More Likely," represented by stars, as is the case when using GCS (example depicted in Figure 6.4(b)). Each of these conditions gauge participant behavior when presented with a proposed AppIntegrity user interface. All five conditions assume the application "Netflix."

Condition *C-CANTV* — verification cannot occur, for instance, due to DNS resolution failure. C-CANTV may indicate an attack (Figure 6.5(a)).

Condition *C-FAIL* — verification process occurred and explicitly failed. C-FAIL likely indicates an attack, but may occur due to an oversight or error by the application publisher (Figure 6.5(b)).

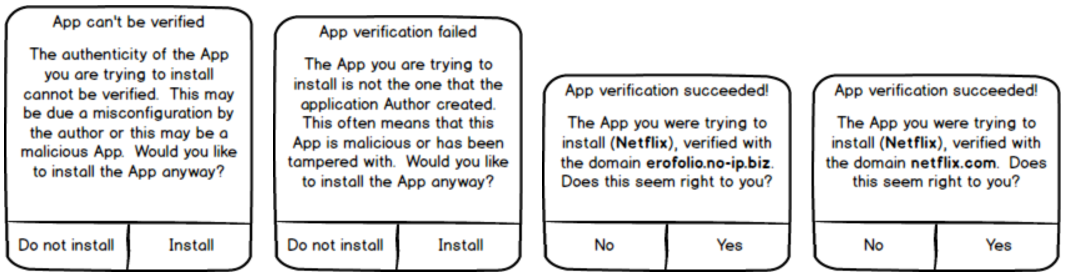
Condition *C-PASSFP* — verification process succeeded, but to the incorrect domain. In fact, the domain presented is the once-malicious domain used in actual fake Netflix Android malware. C-PASSFP is indicative of an attack (Figure 6.5(c)).

Condition *C-PASS* — verification process succeeded, to the correct domain (`netflix.com`). In C-PASS there is no attack (Figure 6.5(d)).

You've just been told you'll get $50 for installing Netflix, how likely are you to install it after seeing this popup? (Condition *C-PASSFP50*). A variation on C-PASSFP where the participant was asked to imagine a $50 USD reward for continuing an install in the case of an application-domain mis-match.

AppIntegrity Interface Study: Results

The majority case for every condition was statistically significant (Wilson score interval 95%), except C-PASS. As shown in Figure 6.2, all surveys presented less than 4% deviation from the expected Internet population (via RMSE)


App can't be verified

The authenticity of the App you are trying to install cannot be verified. This may be due a misconfiguration by the author or this may be a malicious App. Would you like to install the App anyway?

Do not install | Install

App verification failed

The App you are trying to install is not the one that the application Author created. This often means that this App is malicious or has been tampered with. Would you like to install the App anyway?

Do not install | Install

App verification succeeded!

The App you were trying to install (**Netflix**), verified with the domain **erofolio.no-ip.biz**. Does this seem right to you?

No | Yes

App verification succeeded!

The App you were trying to install (**Netflix**), verified with the domain **netflix.com**. Does this seem right to you?

No | Yes

(a) C-CANTV (b) C-FAIL (c) C-PASSFP (d) C-PASS

Figure 6.5: **AppIntegrity UI conditions**: Graphics used in Google Consumer Surveys study.

Condition	Participants	Start	End	Median response time (seconds)	Response Rate	RMSE
C-ROOT	574	2015-09-07	2015-09-09	5.5	73.3%	3.6%
C-ROOT	2004	2015-09-16	2015-09-18	5.6	72.0%	3.5%
C-MARKET	550	2015-09-07	2015-09-09	14.2	70.2%	1.9%
C-CANTV	567	2015-09-07	2015-09-09	18.6	69.1%	2.2%
C-FAIL	599	2015-09-07	2015-09-09	20.1	65.3%	2.0%
C-PASSFP	533	2015-09-07	2015-09-09	21.7	68.6%	2.2%
C-PASSFP50	538	2015-09-07	2015-09-09	23.5	68.1%	2.0%
C-PASS	542	2015-09-07	2015-09-09	21.8	68.1%	2.8%
C-PASS	2000	2015-09-18	2015-09-20	22.6	68.4%	3.7%

Table 6.2: **GCS AppIntegrity efficacy survey results** The answer that the most participants selected in every surveys was statistically significant (Wilson score interval 95%) except C-PASS. Conditions C-ROOT and C-PASS were extended to a larger participant target. For these conditions, results of the original subset and the larger study are both shown. RMSE denotes the weighted average of the difference of prediction population [113] (gender, age, geographic location) and the surveyed sample.

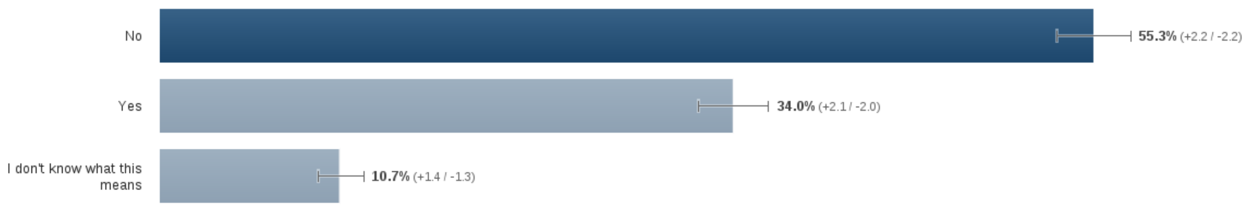


Figure 6.6: **C-ROOT: Results**: Results of GCS condition C-ROOT. The majority answered “no,” however, a substantial proportion answered “yes” to having once rooted an Android device. $N = 2004$.

as collected by the United States Census Bureau [113]. About 7 of 10 people that were offered completed the survey (Response Rate). No surveys presented any GCS “insights” which would be significant differences in response due to age, gender, geography, etc.

Participants completed condition C-ROOT three or four times faster than other conditions. However, this is expected as C-ROOT is essentially a “yes/no” question, while all other conditions required interpreting a graphical prompt with five lines of text or selecting among seven possible answers.

We were surprised by the initial results of C-ROOT where 29.6%(+3.9/-3.6) of users answered reported that they had once rooted an Android device. We therefore increased participants from 500 to 2000, the results of this study are

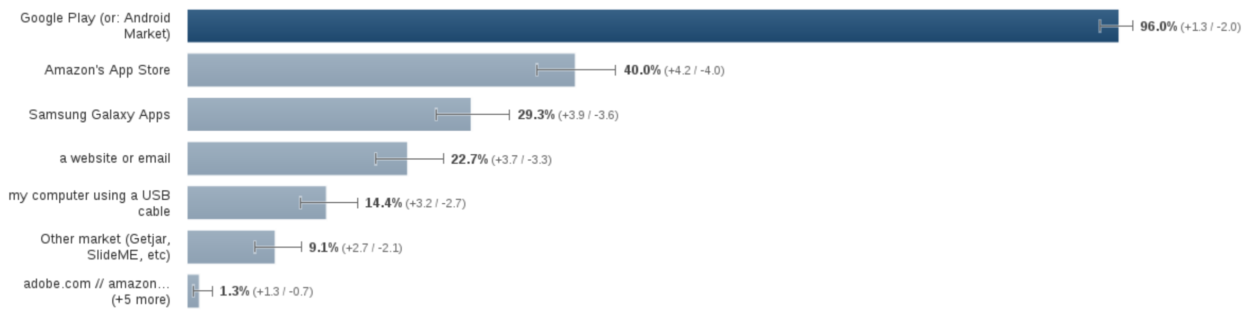


Figure 6.7: **C-MARKET: Results**: Results of GCS condition C-MARKET. The overwhelming majority of participants have used Google Play. However, many participants (at least 40% have used alternative markets). $N = 550$.

shown in Figure 6.6. After 2004 responses, the significant (Wilson score interval 95%) majority, 55.3%(+2.2,-2.2), still reported that 'no' they had not ever rooted an Android device. However, 34.0%(+2.1/-2.0) report that they have rooted an Android device. 10.7% did not know what "rooting" meant.

Upon further consideration, there are many applications available via on Google Play that require a rooted device in order to work. Further, Google Play reports hundreds of thousands installs for many of these applications. For instance SuperSU[6] at 370,000 installs or ROM Manager[7] at 224,000. To put the install size in perspective, the most popular applications (e.g., Facebook) have on the order of 30 million installs, but the install count sharply decreases. Popular applications such as Pandora and Snapchat are each placed at 4 million and Google Docs only at a 300,000, about the same as popular apps requiring root.

One does not have to look far to find other evidence that the percentage of users that root their devices is quite possibly as high as 34%. While our studies were entirely US-based, reported percentages in other countries, such as China, are similarly high at 27.44% [148]. The number is much higher at 80%, when considering devices that cost less than $660 USD [204].

Another possible explanation is the popularity of "alternative ROMs." These alternative ROMs are wholesale replacements for the entire software set on a mobile device. A popular alternative ROM, Cyanogenmod, touts 50 million users [124]. Installing such a ROM requires rooting the device. Further, once installed, Cyanogenmod continues providing root access to the user as a feature of the alternative ROM.

Due to the wording of the question, "have you ever," some number of participant answers might not reflect their primary device. For example, a participant may have rooted a secondary or shared device, a device for some other person, or one time out of curiosity never again to use the device. Similarly, we cannot infer much about the outcome of rooting the device or the participant's disposition toward the outcome.

Condition C-MARKET results clearly indicate that most (96%) users have used Google Play (result is signifi-

[6]SuperSU may be found at `https://play.google.com/store/apps/details?id=eu.chainfire.supersu`.

[7]ROM Manager may be found at `https://play.google.com/store/apps/details?id=com.koushikdutta.rommanager`.

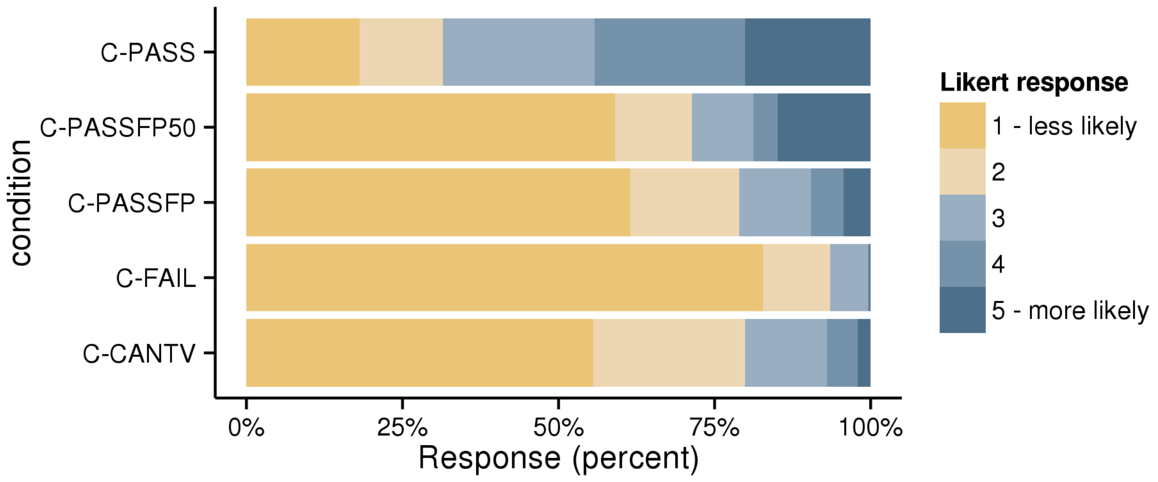


Figure 6.8: **Likert conditions: Result proportions**: Participants were “equally or more likely” (68.5% combined) to continue installation only in the C-PASS condition. In all other conditions, participants were more than 50% “least likely” and often 75% “less likely” (1 and 2 combined). When offered a $50 incentive (C-PASSFP50), the majority still responded least likely, but the proportion of “most likely” more than doubled in comparison to same condition with no incentive (C-PASSFP).

cant; Wilson score interval 95%). However, it is also easy to observe that many users have utilized other markets, highlighting the need for a distributed mechanism working across marketplaces. The most popular alternative markets evaluated in C-MARKET are Amazon’s Appstore and Samsung’s Galaxy Apps store. This result is unsurprising given the vast market-share of Samsung devices and Amazon’s requirements to use the Appstore to take advantage of certain services like Amazon Prime Video. Participants were twice as likely to install applications from e-mail, and 1.5 times as likely to install via USB than other market places. The question verbiage for “other market” may have had an effect, as only two markets were listed in the grouping (Getjar and SlideME) and users may not have mentally grouped other markets into this category. Even so, the free-form text only solicited seven answers from participants: adobe.com, amazon, backup from cloud storage, firefox, humble store, personally made, and “not sure though it came with my phone.” Clearly, the “amazon” answer may have been a misunderstanding. The last free input is possibly a reference to Samsung Galaxy Apps store, but it could also be any number of stores that manufactures and cellular providers pre-install on devices.

In all the remaining conditions the participant was asked to gauge how likely they were to install an application after being presented with a modal dialog. Each of these questions gauge participant behavior when presented with a proposed AppIntegrity interface. Since the answers were based on a 5-point Likert scale from “Less Likely” to “More Likely,” a middle response indicates that the participant is no more or less likely to install due to the dialog.

The five-star Likert scale is reported numerically, with the leftmost (labeled “less likely” in Figure 6.4) being 1 and the rightmost 5. Likert results for each condition (Figure 6.8) reflect positively on the intended operation of

AppIntegrity, particularly in the situations that present risk to the user. Conditions other than C-PASS, all exhibit significant (Wilson score interval 95%) "least likely" responses. Combining the "less likely" responses (i.e., 1 and 2) for each condition, C-CANTV, C-FAIL, and C-PASSFP are all over 75%. The increased deterrence is promising as the four conditions where 75% of participants are less likely to continue installation are those that present harm to the user.

In the paired conditions C-PASSFP and C-PASSFP50, when offered \$50, a full 10% of participants are taken from each other answer category into the most likely. The two values that change the most are 2, and 3. So, those that were most concerned about the app, remained so (62 vs 59 %). However, more of those that were concerned, but not as much—value 2 (27 vs 12 %) and value 3 (22 vs 10 %) shifted toward the most likely to install (4 vs 15 %). So, 10% of participants, all of whom would otherwise not install the application, can be incentivized to due so via a \$50 payment.

C-PASS is the only condition in which a statistically significant winner was not observed. In C-PASS, answers 3 and 4 are statistically similar. With answer 3, equally likely, at 24.3%(+1.9/-1.8) and answer 4, slightly more likely, at 24.1%(+1.9/-1.7) no winner could be determined. Even so, the result is positive regarding user behavior. Users were equally or slightly more likely to install when AppIntegrity correctly verified, and significantly less likely to install otherwise. Somewhat perplexing is the least likely case for C-PASS. More participants 18.1%(+1.7/-1.6) selected 1, than 2 at 13.4%(+1.6/-1.4). This reluctance to install may simply be due to discouragement inherent in presenting an unfamiliar security dialog, or perhaps participant that did not recognize or desire the "Netflix" application. The reluctance to install in the C-PASS condition is concerning, because a substantial number of participants are dissuaded from installing applications that are quite likely safe, and, in the absence of AppIntegrity, would be installed by users.

6.4.2 Domain Recognition Study

In the two cases where verification succeeded for Netflix, in one the success has occurred to a malicious domain (C-PASSFP), in the other it occurred to the legitimate domain (C-PASS). This situation is akin to domain recognition and deception, general limitations that are likely to persist independent of AppIntegrity implementation details. Therefore, we investigated these issues more deeply.

Unlike the typical uses-cases for domain recognition and deception, such as the Address Bar in a browser, AppIntegrity requires the user to pair two pieces of information together: a domain and an application name. We hypothesize that this more complex task will pose a greater burden upon participants and therefore expect relatively poor performance. Further, since AppIntegrity users are not typing URLs, but instead verifying a displayed URL, our problem is more one of recognition than typosquatting.

To examine this recognition ability we considered five additional conditions expanding upon the two pairs (Netflix:`netflix.com` and Netflix:`erofolio.no-ip.biz`) already studied. We studied 20 instances of each of the five conditions, creating a total of 100 application-domain pairs. The precise 100 pairs we evaluated are shown in Table 6.3. The study was implemented by presenting five pairs to each of at least 150 participants in a GCS survey (20 surveys total), from November 9 to 11, 2015. Each pair was presented graphically exactly as in Figures 6.5(c) and 6.5(d), only varying the name of the application and the domain. Each survey question contained one pair from each condition, allocated randomly. The order of the five pairs in each survey was also random.

Condition *C-REAL* presents application names paired with the real domain, derived from current application package name, regardless of how well the domain appears to match the application. C-REAL allows us to consider if AppIntegrity could be useful immediately, with real applications as they exist today. We hypothesize that users can match applications often, but not universally. For example, one may expect the Facebook application to be matched with `facebook.com`, but one may not expect the Tango application to be matched with `sgiggle.com`.

Condition *C-IDEAL* presents application names paired with an ideal domain, as subjectively determined by the authors. This condition represents the reality that might exist after effecting changes to address discrepancies in C-REAL. Continuing the example in C-REAL, the Tango application would be paired with `tango.com` instead of `sgiggle.com`. As this condition is ideal, one might expect participants to perform particularly well in this condition.

In both C-REAL and C-IDEAL, the desire is for participants to select mostly positive Likert responses, middle or higher. Middle is acceptable for the purposes of AppIntegrity as the user would already be attempting to install the application at the time of the dialog prompt. A survey participant selecting the middle response is “equally likely” to continue as before seeing the dialog, so the intent to install persists.

In the next three conditions, the desire is for participants to select mostly negative responses. In these cases, the AppIntegrity prompt has indications of potential harm that the participant may detect.

Condition *C-RECO* presents application names paired with a real domain, but not the domain associated with the application. In this case, a participant may recognize the domain name and mis-place trust. In practice, the situation represented in this condition is unlikely, as the owners of recognizable domains are not likely to mount attacks against other applications. An instance of this condition might be the Facebook application matched with `google.com`.

Condition *C-TYPO* presents the “typojacking”-style attacks on domain recognition. The domains presented in this condition are intentionally misleading. Given the poor historical performance [159], one may expect poor results from participants in this condition. An example of C-TYPO, might be a Google Photo’s application paired with `goog1e.com` (where the expected lowercase “L” is actually the number one). For C-TYPO, minor, visually similar, variations were made to domain names from C-REAL.

Condition *C-MALW* presents application names paired with domains that the authors consider obviously bad. These names are largely taken from real malware Domain name Generator Algorithms (DGAs) with a few additional characters added to prevent participant risk, or ones that might be registered specifically as part of an attack campaign. Domain names in C-MALW were created solely for this study, and each was tested prior to the study to ensure that

Application Name	C-REAL	C-RECO	C-IDEAL	C-TYPO	C-MALW	Selection Set
Facebook	facebook.com	wikipedia.org		facebook.org	homeip.net	Top10 free
Google Photos	google.com	twitter.com		goog1e.com	portella.ru	Top10 free
snapchat	snapchat.com	craigslist.org		snap-chat.com	larr.us.com	Top10 free
Pandora Radio	pandora.com	yahoo.com		pand0ra.com	arjutta44.com	Top10 free
Instagram	instagram.com	amazon.com		1nstagram.com	platitat.pl	Top10 free
Twitter	twitter.com	lowes.com		twittter.com	rwyoehbkhdhbb.info	Top100 free
360—Security Antivirus Boot	qihoo.com	kohls.com	360security.com	qhoo.com	xxx-filez.de	Top100 free
ZEDGE Rintones and Wallpapers	zedge.net	wikia.com	zedge.com	zegde.com	2015mmmmmm.cn	Top100 free
Tango—Free Video call & Chat	sgiggle.com	live.com	tango.com	sgiggles.com	yeiesmomgeso.org	Top100 free
Solitare	kathleenoswald.com	usps.com	hasbro.com solitarecards.com	katheleenosald.com	wwwtariff.com	Top100 free
Geometry Dash	robtopx.com	yahoo.com	geometrydash.com	robotpx.com	9u4.cn	Top10 paid
True Skate	trueaxis.com	amazon.com	trueskate.com	true-axis.com	yeuqiik.com	Top10 paid
Minecraft: Pocket Edition	mojang.com	craigslist.org	minecraft.com minecraft.net	mo-jang.com	oumaacc.com	Top10 paid
Sleep Cycle Alarm Clock	northcube.com	go.com	sleepclock.com	northcubee.com	00000007.ru	Top10 paid
Power Ping Pong	chillingo.com	wikipedia.org	powerpingpong.com ppp.com	chilingo.com	Club-446.com.ar	Top10 paid
Call of Duty Black Ops Zombies	atv.com	pornhub.org	callofduty.com activision.com	aatv.com	njr76ail.rr.nu	Top100 paid
Grand Theft Auto: Vice City	rockstargame.com	office365.com	gta.com vicecity.com	rockstargamer.com	b8t.at	Top100 paid
Ackinator the Genie	digidust.com	bleacherreport.com	ackinatorgames.com atg.com	digitdust.com	peskolastruikaz.com	Top100 paid
WolframAlpha	wolfram.com	salesforce.com	wolframalpha.com	wo1fram.com	nt88.in	Top100 paid
Five Nights at Freddy's 3	scottgames.com	bbc.com	fivenightsatfreddys.com	scottgmase.com	zljvtovlzobhpbjrnpbdatzx.org	Top100 paid

Table 6.3: **Application-domain pair recognition conditions.** Each of Applications were paired with a domain name from each of the other columns for App-Integrity user studies. For the C-IDEAL condition, several applications already employed the ideal domain, therefore six applications have two C-IDEAL domains to make 100 pairs.

the domain did not actually resolve. Each domain name was inspired by real malicious domains as found in the SANS "High Sensitivity Level" suspicious domains list.[8] An example of a C-MALW pair might be the application Instagram with domain `adfhadfiubknm.com`.

More popular applications and domains are arguably more recognizable. So, for each of the five additional conditions we tested highly popular applications and less popular applications. To this end, we randomly selected five applications from each of the following Google Play sets: 10 most popular free, 100 most popular free, 10 most popular paid and 100 most popular paid.

In our random selection, we considered the real domain for the application to also be the ideal case for six pairs, that is, sets C-REAL and C-IDEAL intersected. For instance, Twitter already uses the ideal domain: `twitter.com`. Instead of repeating the condition twice for each of these six pairs, we created additional conditions for other applications that plausibly had more that one ideal domain. In this way, 20 C-IDEAL pairs were evaluated.

Additionally, two domains were too short to effectively test typojacking (`wb.com` and `ea.com`). For this reason, Scribblenauts Remix (10 most popular paid) and Monopoly Millionaire (100 most popular paid) were replaced with new random draws: Power Ping Pong and WolframAlpha.

Domain Recognition Study: Results

A total of 3042 responses were recorded across all 20 surveys. Each survey contained five questions (one application name:domain pair each) for a total of 15,210 pairs presented to participants. Each survey contained 150 to 158 responses, since there are less than 250 responses, RMSE is not reported.

The median response time was 24.0 seconds on the initial question indicating that participants took some time to understand the question. Each of the following four questions were answered progressively faster as participants became acquainted with the question format. The median time for the second question was less than half at 11.4 seconds. The trend continued, albeit not at drastically, with the third (7.9 seconds), fourth (6.5 seconds) and fifth (5.5 seconds) questions.

For all surveys, the five-star Likert scale is reported numerically, with the leftmost (labeled "less likely" in Figure 6.4) being 1 and the rightmost 5. Of the 3042 responses, 681 comprised of all five survey answers being the same, that is, the participant selected the same value for all five pairs. We questioned whether this might indicate that the participant is attempting to complete the survey disingenuously in order to receive the reward. Of these 681, the overwhelming majority (467) selected one star. Eighty-eight participants selected all two star responses, 71 selected three star, 22 selected four star and 33 participants selected five stars for every responses. When looking at all 681 responses, but holding the 467 one-star responses to extra scrutiny, we see that they are all likely valid responses. The z-score for response time in any question only indicated a possible outlier in 46 cases, 19 of which were on the initial

[8]List may be found at `https://isc.sans.edu/feeds/suspiciousdomains_High.txt`.

Survey	Participants	Start	End
1	151	2015-09-09	2015-09-10
2	150	2015-09-09	2015-09-10
3	150	2015-09-09	2015-09-10
4	152	2015-09-09	2015-09-10
5	151	2015-09-09	2015-09-10
6	152	2015-09-09	2015-09-10
7	150	2015-09-09	2015-09-10
8	158	2015-09-09	2015-09-10
9	150	2015-09-09	2015-09-10
10	150	2015-09-09	2015-09-10
11	152	2015-09-09	2015-09-10
12	156	2015-09-09	2015-09-10
13	150	2015-09-09	2015-09-10
14	153	2015-09-09	2015-09-10
15	150	2015-09-09	2015-09-10
16	151	2015-09-09	2015-09-10
17	163	2015-09-09	2015-09-10
18	151	2015-09-09	2015-09-10
19	151	2015-09-09	2015-09-10
20	151	2015-09-09	2015-09-10

Table 6.4: **GCS AppIntegrity domain recognition survey results** Each survey was specified to have a minimum of 150 responses. Unlike Table 6.2, RMSE is not reported because the sample sizes are smaller than 250.

question. In no case did any participant exhibit a possible outlier response time on more that one question. In 11 cases, a participant took longer than 30 minutes to answer a question, but had normal response times for the other 4 questions. Possibly, the participant was interrupted (e.g., phone call or text message) mid-survey, but still completed all five questions. Significantly, of the 681, 391 were female whereas only 288 were male ((χ^2 = 19.571, df = 1, p < 0.001)

The average response proportion for each condition is shown in Figure 6.9. A "negative response" is 2 stars or less, while a "positive response" is 3 stars or more. In the three cases were negative responses were desired, we can see sharp decreases (compared to positive responses) in 4- and 5-star responses, as well as sharp increases in 1- and 2-star responses. This result is promising for AppIntegrity as it demonstrates that participants were further deterred from installing the application in every potentially malicious case. Participants were particularly good at recognizing C-MALW (82% negative response) and C-RECO (71% negative). The participants in our study did not mis-place trust to the recognizable names, on the contrary, they were actually more discouraged by mis-matched pairs than by the misleading names in C-TYPO. C-TYPO was the worst performing of the conditions where negative responses were desired. Even so, participants performed well with, 61% providing a negative C-TYPO response.

Ultimately, our hypothesis for C-REAL is confirmed; participants were able to often match applications correctly, but not universally. Only 47% of participants selected a positive response (3-5 stars). So, if AppIntegrity were to be implemented with applications as they exists today, more than half of users may be discouraged from installing application-domain pairs studied in this survey. While not measured in this study, we might theorize that users were pre-disposed to being discouraged, as they are discouraged by all security-related warnings [90, 92]. The Top10Free

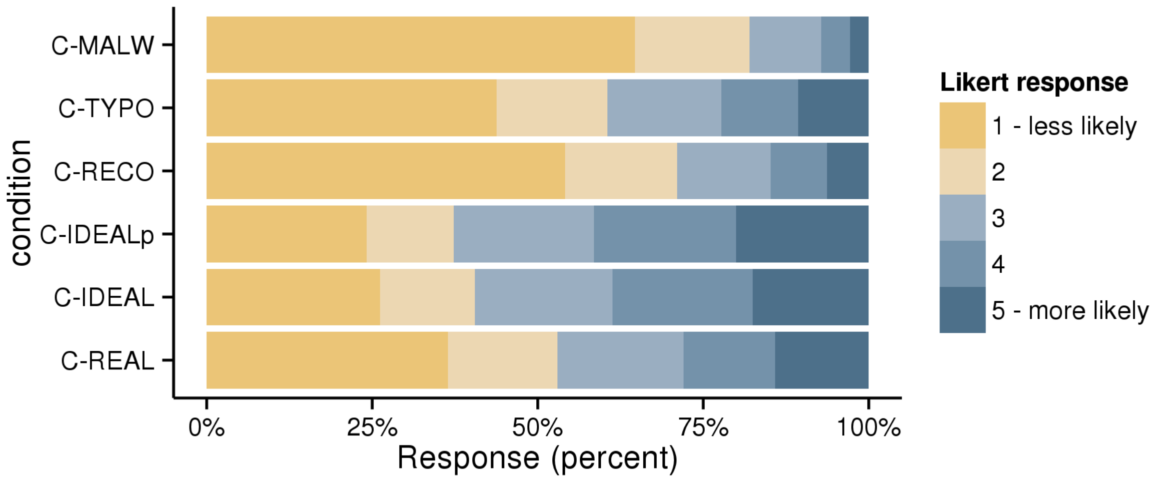


Figure 6.9: **Application name-domain understanding, by condition**: Participant responses indicate that AppIntegrity is able to provide safety, particularly in negative cases. In the positive conditions (C-REAL and C-IDEAL) approximately half of the participants were "at least as likely" to continue installing the application. Unfortunately, in C-REAL, 53% were "less likely" to install an application after seeing an AppIntegrity prompt. In the negative conditions (C-MALW, C-TYPO, and C-RECO), the majority of users were dissuaded from continuing the install. This dissuasion was particularly true in C-MALW and C-TYPO where more than 3 of 4 participants were "less likely" to install following an AppIntegrity prompt. C-IDEALp, shows C-IDEAL augmented with the Top10Free results from C-REAL, which already employed the ideal domain names.

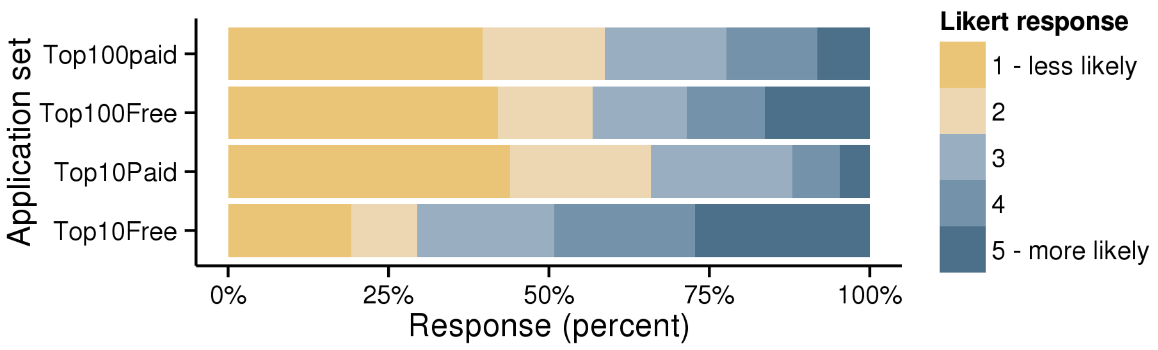


Figure 6.10: **Application name-domain understanding, C-REAL by selection**: Real domains are most recognizable in top 10 free. In particular, paid applications tend to be published by companies that use corporate domains that are not representative of the application name. Example: Tango paired with `sgiggle.com`.

result for C-REAL (which we also consider to be IDEAL) also closely matches the prior result of C-PASS (see section 6.4.1), discounting concerns that C-PASS might have been biased due to using a single application (Netflix) and domain (`netflix.com`).

Also encouraging is that participants were able to match applications twice as well for Free applications than Paid, as shown in Figure 6.10. The better performance in Free applications is encouraging because free applications are likely much more popular than paid applications, meaning AppIntegrity may work better in the common case. For

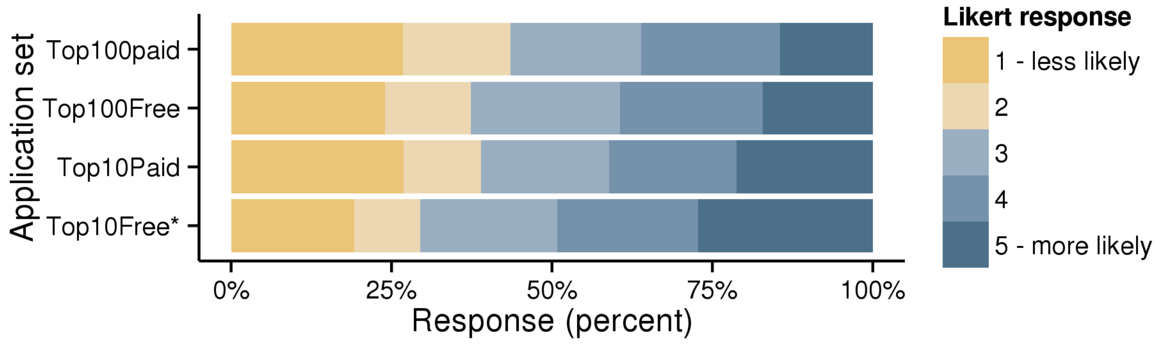


Figure 6.11: **Application name-domain understanding, C-IDEAL by selection**: C-IDEAL is the best performing condition in all categories of application. Participant ability to recognize ideal pairings is promising for AppIntegrity because application publishers could move toward this ideal condition with simple changes to mobile applications. As in other conditions, the more popular applications perform better. However, in C-IDEAL, all four categories perform well relative to other conditions—well over half of participants respond positively to ideal pairings. Example: Tango paired with `tango.com`. Note: The Top10Free values are not shown in Table 6.3 because the same pairs were already surveyed as part of C-REAL.

instance, using the minimum reported download counts for Google Play (only ranges are reported), the top ten free applications are downloaded between 10 and 5000 times more than the top ten free. On average, the top ten free applications are downloaded 195 times more than paid.

The ideal pairing (Figure 6.11) performed better than C-REAL in all positive cases (60% 3-5 stars). The result is inspiring as the desired improvement is relatively straightforward for application authors. Application authors are often already domain owners, in which case a minor application software modification could yield better AppIntegrity performance. Of course, such changes may also cause conflict if the name of an application happens to coincide with a domain owned by some other party. The better performance of C-IDEAL can be measured in spite of the ideal domain for the top six applications not being tested (as the real domain in-use was already the ideal domain). So the C-IDEAL numbers could be augmented with C-REAL values in those cases. The augmentation can be observed in Figure 6.11, where the Top10Free values are the same as those in C-REAL (Figure 6.10).

Unlike the first two conditions, in the remaining conditions (C-RECO, C-TYPO, and C-MALW) the desire is for participants to halt the installation process—to provide a “less likely” Likert response. In these cases, the AppIntegrity prompt is meant to protect the users from harm. Optimal responses would have resulted in Figures 6.12, 6.13, and 6.14 all depicting 100% one or two star Likert responses.

When prompting recognizable domains with incorrect applications (C-RECO), more than half (54.1%) of participants were “least likely” to continue installation. Further, 71% of participants responded “less likely” to continue (selecting one or two stars). Overall, the participants did not misplace trust when prompted with a familiar, but mis-

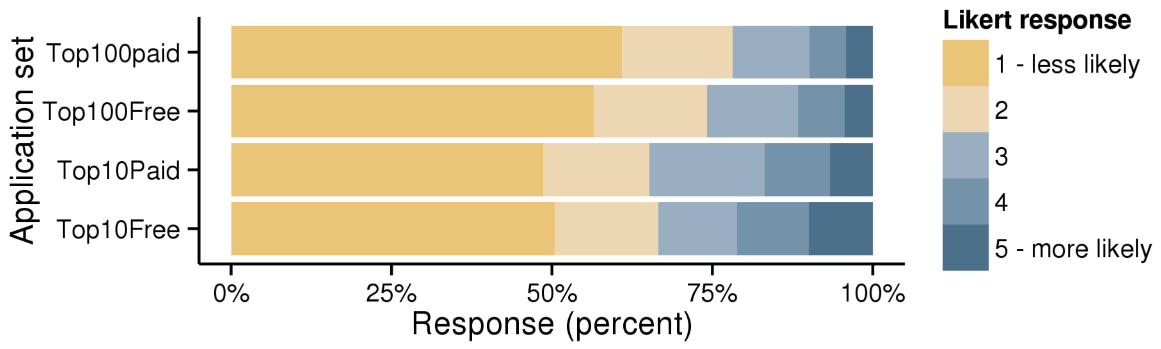


Figure 6.12: **Application name-domain understanding, C-RECO by selection**: In all categories of application, participants were dissuaded from installing applications matched with recognizable but incorrectly paired domains. On average, more than half (54.1%) of the participants were "least likely" (1 star) to continue an install following the AppIntegrity prompt. Example: Facebook paired with `google.com`.

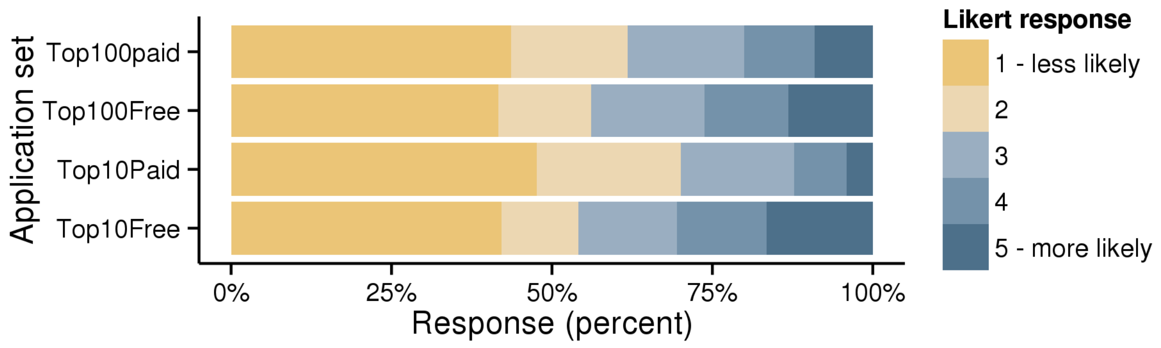


Figure 6.13: **Application name-domain understanding, C-TYPO by selection**: Counter-intuitively, participants were generally able to spot small discrepancies in domain names. Example: Google Maps paired with `goog1e.com`.

matched, domain. Figure 6.12 shows overwhelming proportion of one and two star answers, however, nearly 30% of participants would still continue installation.

The C-TYPO results presented in Figure 6.13, show a slightly different distribution among the category of applications than C-RECO in Figure 6.12. Overall, C-RECO performs better (71.0% negative response) compared C-TYPO (60.6%). However, for the top ten paid applications, C-TYPO performs slightly better at 70.2% negative versus 65.2% in C-RECO. In all other types of applications, C-RECO clearly performs better.

While far from perfect, and somewhat contrary to related work, our participants were generally able to identify and correctly respond to typojacking style domains. Given the dire past performance in related research, we expected the more complicated task of reasoning about the application name in addition to identifying the misleading domain would result a poor outcome. However, it seems that pairing information (the application name) has potentially added context, aiding the participant in recognizing the typojacking domain. However, the precise reason for the

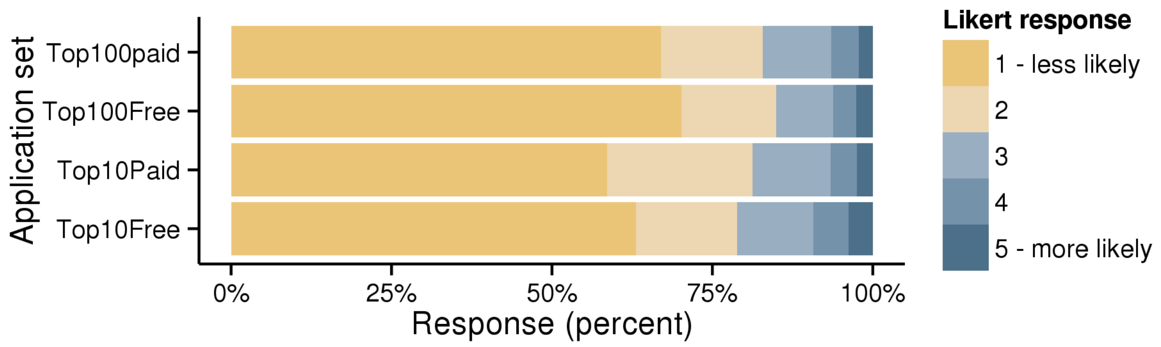


Figure 6.14: **Application name-domain understanding, C-MALW by selection**: AppIntegrity prompts were highly effective in condition C-MALW. Here the overwhelming majority (82.0%) of participants were dissuaded from install applications pairs with apparently malicious domains. Example: Instagram paired with `adfhadfiubknm.com`.

unexpectedly positive performance cannot be discerned from our data. Users may have improved at this task over time, Android users might be suited particularly well at such recognition, or any number of other possibilities. Purely from the perspective of AppIntegrity, the C-TYPO result is promising in that AppIntegrity can likely help protect users from harm due to repackaging attacks that aim to deceive the user with a typojacking style domain.

Undoubtedly, C-MALW was the best performing condition. A full 82.0% of participants were dissuaded from continuing an application install (64.7% were “least likely” to continue). In the case of C-MALW, AppIntegrity could certainly provide safety to a substantial proportion of participants. Unfortunately, even C-MALW did not exhibit perfect results. Eighteen percent responded positively, and even 2.8% responded “most likely” (5 stars) to proceed.

6.4.3 Observations from user studies

Users are seemingly generally able to identify and theoretically act upon proposed AppIntegrity feedback. In particular, AppIntegrity appears to work well in the situations where AppIntegrity is protecting users from harm—even when participants are offered compensation to disregard security advice. The users were particularly dissuaded from apparently malicious URLs and applications incorrectly paired with recognizable domains. However, our user studies are relatively small compared to the Android user base, and the ecological validity of the AppIntegrity interfaces remains an open question.

Surprisingly, a substantial proportion of users have experience in rooting smartphones. Perhaps, even if Google, AOSP, Amazon or other large players to not implement AppIntegrity, ex post facto deployment might be possible on rooted devices. The large proportion of users that have once rooted a device, give hope that such a deployment may well be able to protect many users.

In the end, the AppIntegrity protocol along with the proposed UIs should protect users in many repackaging attacks (which represents approximately 73% of mobile malware [225]). Further studies should be performed to determine the effects of domain or brand recognition on paired recognition as required by AppIntegrity.

As it stands now, it seems unlikely that today's leading markets would encourage AppIntegrity use as part of every application installation. In particular, in the positive conditions of Figure 6.9 (C-IDEAL, C-IDEALp and C-REAL), a substantial proportion of users were dissuaded from continuing application installation (>25% least likely to continue). Ultimately, the determination of an acceptable false-positive rate rests with each entity (including market proprietors). Further, the process of determining each rate will vary widely, but almost certainly requires economic risk analysis and is left for future work. So, while Figure 6.9 supports the conclusion of better overall safety, it also may lessen overall use of the marketplace. For this reason, it is possible that better overall protection might be realized by taking decisions out of the user's hands when possible. Based on our survey data, one option might be to present AppIntegrity UIs only in certain circumstances.

6.5 Related Work

Spoofing attacks similar to the Netflix malware were theorized by Felt et al. in [110]. Felt et al.'s work [110] predates the recent Netflix spoofing malware which very closely mimics the Facebook attack described in their work. Additionally Felt et al. also provide a survey of much of the mobile malware discovered from 2009 to 2011 in [109].

In prior work [230], we observed application repackaging as one type of an "unprivileged attack." The class of unprivileged attacks is one part of a greater taxonomy targeting Android devices. We also observe that malware is often present in alternative ("black") markets and such applications often "offer no additional value to the consumer." These observations are confirmed in our alternative market corpus discussed in section 3.3. Repackaging detection (or at least an indication) may be discussed as a factor of application size in relation to the original application. That is the addition of malicious software and re-signing process may add an observable, nearly constant size to the original application. While the larger size is not technically required, measurements do support this assertion [144]. Both our work and that of Burguera et al. [69] observe that current signing processes in no way inhibit repackaging and republication of applications.

In conjunction with alternative market malware research, Zhou et al. present a new tool for re-actively detecting malware found in markets [249]. In contrast, AppIntegrity strives to proactively prevent the installation of software signed by those other than the originator.

Chen et al. [76] use application metadata to identify web applications which the authors then provide a means of app isolation. Chen et al. reference the Chrome Web Store which allows "verified apps." The procedure for obtaining the "verified" icon in the store is to pay a $5 fee and the application developer must verify domain ownership via Google's "webmaster tools." The term "verified" is used differently here, as the verification is proven to the market which then assures the consumer. The additional assurance provided by proving domain ownership is likely useful as a means to increase application use and market reputation, but is somewhat orthogonal to the end-to-end integrity provided by AppIntegrity.

Enck et al. describe a lightweight application certification service, Kirin [99]. This service forces applications to pass several rules at install-time, such as the absence of permission combinations the rule creator deemed dangerous. AppIntegrity could possibly be implemented as a feature of Kirin, or independently as described above, in addition to Kirin.

AppIntegrity focuses on ensuring end-to-end integrity for applications, and makes no attempt to analyze the inner workings of an application or otherwise protect the user from applications that are malicious from origin. For this reason it makes sense to pair AppIntegrity with taint tracking systems such as TaintDroid [97] or PiOS [94] in order to detect privacy leaks. Similarly, Hornyack et. al have retrofitted Android [128] in a way that permits executing existing applications in a safe way.

6.6 Future Work

Our GCS participant pool consisted entirely of US participants. We expect that some survey questions, particularly C-ROOT and C-MARKET may exhibit varying results when using an international participant pool. For example, C-MARKET clearly indicates that the top markets are Google Play, Amazon's Appstore and Samsung Galaxy Apps. Worldwide, we might expect Google Play to be unpopular in China, participants instead favoring Myapp or Baidu. Now, GCS supports nine additional countries for Android users: United States, United Kingdom, Canada, Australia, Germany, Italy, Netherlands, Mexico, Brazil, and Japan. While not a fully representative of worldwide participants, countries could be compared pairwise, and we hypothesis that responses would be different. However, we expect the observations with respect to AppIntegrity to remain similar.

A substantial proportion of users have some experience in rooting smartphones. For some future work, such as an implementation of AppIntegrity meant to work on rooted devices, deeper study about the prevalence and nature of rooting is warranted. Utmost, is understanding if a participant's primary and current device are in a rooted state and if the participant uses an alternative ROM. If participants had rooted devices in the past, as an experiment, without success, etc, then an AppIntegrity deployment that relies upon rooting is unlikely to provide safety to many users.

Additional survey data could be collected for many variations and extensions of our AppIntegrity evaluation. We suggest that further insight into domain or brand recognition may prove useful in designing UIs or minimizing the set of situations that result in an AppIntegrity prompt. Similarly, our surveys only studied one suggested UI. Other formats for prompts could be evaluated or even inclusion of additional information, such as displaying the publisher name in additional to the domain and application name (which may help alleviate some mis-matching where commercial entities publish many applications using a domain based on publisher name, not application name).

In addition to collecting additional survey data, the protocol itself could be extended. Better user protection is likely attainable via integration of crowdsourcing techniques with AppIntegrity. For instance, Protect My Privacy [53] presents users with application resource usage paired with crowdsourced opinion of the application. With Protect My Privacy, users of rooted iOS devices may initially have concern that an application has the ability to query location data, but be re-assured that most users "approve" of the application.

Similarly, other crowdsourced applications could be layered upon AppIntegrity, such as Perspectives [237]. Perspectives essentially ignores the CA component of the Secure Sockets Layer (SSL) PKI, effectively using PKCS component similar to AppIntegrity. A decentralized network of servers allows users to place trust into parties that the users selects instead of parties the website operator selects (e.g., a CA). In the case of AppIntegrity, the applications are already assumed to be distributed in a decentralized way, so it follows that a decentralized trust model is technologically acceptable, and it may well provide better overall safety. As an additional benefit, a system like Perspectives could be implemented quickly, without requiring action on the behalf of publishers (e.g., publishing an AppIntegrity key to DNS).

A key research component of our user studies revolved around any trust implicitly granted to an application-domain pair due to recognizable domains. AppIntegrity could be combined with other technologies that focus on domain-level security. For example, domain reputation or brand monitoring services could be employed in conjunction with AppIntegrity to assist users in domain recognition and potentially misleading situations. Similarly, anti-phishing services could be employed to assist users in identifying potentially harmful domains.

AppIntegrity relies upon public keys being readily available and bound to an entity via domain ownership. The availability of these keys could complement other application market functions such as application revocation. Currently when malware is identified in Google Play, both the publisher and the consumer are at the mercy of Google to remotely uninstall applications from infected devices. By extending our presented protocol to verify applications prior to execution, disabling of malware can be performed by either the market proprietor (e.g., Google) or by the domain owner (e.g., the publisher).

6.7 Limitations

To provide the best protections on the device, the AppIntegrity protocol should be performed as part of the install process. Therefore, to provide the best protection, Android must be modified. The most straight-forward modifications are to natively include AppIntegrity as part of the package manager or to extend the Android APIs to allow administrative applications to enhance the application install process. Until Android is modified to facilitate this form of protection, AppIntegrity can still fully protect users of rooted devices (possibly one of every three users, based on our study; see Figure 6.6), and provide reduced risk via periodic "scanning" on non-rooted devices.

While repackaging is observed in the majority of mobile malware, the technique itself is not necessarily malicious. Consider "benevolent repackaging" where a third party appends beneficial code segments to existing applications. As with repackaged malware, this beneficial code is also not software that the developer intended for users. For example, consider a third party that has translated an application only available in English to Spanish. The resulting application might be published, unbeknownst to the developer, and subsequently used by Spanish-speaking users that otherwise would not use the application. In some cases, the developer might be amicable, in others ambivalent or bitter. As presented in this chapter, AppIntegrity ensures that the application installed by the user is that which the developer intended. As such, application translations would need to be provided to the original developer, who would then need processes and policies for incorporating the translation into the application — thereby signing the application with the developers key and enabling AppIntegrity.

The only user-facing efficacy studies we performed were conducted using GCS which may introduce some undesired bias. Further, we only surveyed U.S. participants, which might introduce bias across all conditions. In particular, we expect the answers in the C-MARKET condition to vary based on geographic area. We expect other alternative markets to have considerable market share. For example, in China, we expect Google Play to have a relatively low proportion, while Qihoo 360,[9] MyApp,[10] Baidu,[11] and MiUi,[12] to have considerable proportions.

GCS participants were clearly subject to some priming. We attempted to mitigate this influence by randomly ordering questions for each condition. However, we can still observe the effects of priming in that participants answered each of the five questions progressively faster.

We did not obtain any additional detail regarding the 34% of participants that have rooted an Android device. For example, if their rooting experience was a success, if the participant ever (or still) primarily used a device that was rooted, how many devices have been rooted or how often.

Many of the domains listed in Table 6.3 were devised by the authors and therefore may introduce bias or inaccurately represent the desired condition due to the subjectivity introduced by the authors . For instance, some might dispute that the ideal domain for the "Solitaire" application may be "hasbro.com." The C-MALW and C-TYPO domains are equally subjective as the malicious domains started as actual malicious domains, but were slightly modified to protect our participants from inadvertently visiting dangerous domains.

As we were mostly concerned with whether a participant would continue the install process following an AppIntegrity prompt, arguments could be made toward implementing the questions with only "yes" and "no" options (vs

[9]`http://360.cn`
[10]`http://myapp.com`
[11]`http://shouji.baidu.com`
[12]`http://miui.com`

Likert). Selection of such components of study design are subjective. We elected to use Likert in order to observe the proportions of likelihood that participants exhibit.

The approximately 25% negative response during the ideal conditions (C-PASS, C-IDEALp) implies that it is unlikely that protocol adoption would be embraced by markets, OS providers, developers, etc. As shown earlier in this chapter, adoption of AppIntegrity protects users from risk, but users also do not install correctly verified applications about one time in four — a rate that is likely too high for these parties to endure.

In all the GCS studies, we only used one format of potential UI display. There is considerable work in UI from which we based our design. However, there are many variations in this still-evolving field that may produce different results in studies similar to ours. Larger buttons, incorporating color, alternate layouts, nudging the user toward a preferred selection, and many more variations could be studied.

One aspect of user recognition not fully addressed in our studies is that of diminishing recognition as popularity decreases. Our studies offer a step in the right direction by differentiating top 10 vs top 100 applications. However, it remains unclear exactly how user recognition diminishes in relation to application popularity. Our studies do indicate that AppIntegrity can provide safety for users of popular applications, precisely those that are likely to be targeted by miscreants in repackaging attacks.

6.8 Discussion

In Chapter 3 we demonstrated that some mobile application markets distribute malware almost exclusively. Most of this malware is *repackaged* in some way giving victims something desirable to execute—the features of the original application.

In order to mitigate the threat of repackaging, we present an end-to-end verification protocol, AppIntegrity, that facilitates cryptographic verification between the software creator and the end consumer. Even though our reference implementation and related discussions largely focus on Android, AppIntegrity can leverage existing application signing for many mobile platforms, and indeed application delivery mechanisms for traditional PCs. The most common mobile consumer platforms: Android, iOS, and Symbian all already use application signing in some way, and can benefit from a verification system like AppIntegrity. By binding public keys based on domain ownership, AppIntegrity has the ability to leverage PKCS without the need for a complicated PKI, further contributing to making AppIntegrity rapidly deployable.

The cost of adoption for AppIntegrity is very low. The minimal network and local resource use is ideal for the constrained environment of mobile devices. Furthermore, the end-to-end protocol can be used with existing official and alternative markets alike. Applications that are republished via alternative markets can be downloaded and verified by a user who can be confident that the installed software is what the developer intended for delivery.

Similarly, AppIntegrity would be compatible with "private markets" given that the devices that have the private markets provisioned have network access to protected domain spaces. Consider a "secure Android" under development by a government entity, as long as devices can access the government network (via VPN for example), certificates can still be retrieved from the appropriate URLs and verification can be performed.

Relating to Android in particular, AppIntegrity requires no changes to the existing Android development process. The application structure and cryptographic signing are used in exactly the same manner as currently employed. Similarly, AppIntegrity is designed to make use of the self-signed keys widely used by Android developers. Minimal changes to the Android framework could enhance the ability for AppIntegrity protect users, but even when used with the current version of Android, AppIntegrity can provide added safety by rapidly uninstalling unverified applications, and providing building blocks for future protocols and services.

We hope that developers elect to publish their public keys as we describe in section 6.2.1. For the, potentially large, proportion of users that possess rooted devices, AppIntegrity could be employed without any marketplace imposition placed upon developers. In order to encourage adoption, we hope that Google will adjust the Android developer documentation and effectively make public key publication part of the standard developer account setup.

AppIntegrity provides a foundation for providing safety at many junctures of online software dissemination. Those publishing software can safely disseminate via many marketplaces by proactively making their public signing key available. At the same time, users receive assurance that the software being installed to their smartphones is that which the publisher intended.

Chapter 7

Conclusions

Malware is clearly afflicting Android devices despite designing Android with security in mind. As with PC counterparts, mobile malware presents many gradations of technological and criminal sophistication. However, mobile devices present new avenues for miscreants to attack. Avenues that are actively pursued, lending to a tremendous growth in mobile malware.

One relatively new phenomenon is that of online software dissemination, the primary mechanism in which the base functionality of a smartphone is extended. Through online marketplaces, users of PCs or smartphones may locate and install applications. Market proprietors, smartphone manufacturers and smartphone operating system creators all create software dissemination policies and procedures meant to attract and retain users. Indeed, single entities often adopt several of these roles in a bid to maintain total control over then entire chain. For instance, Apple manufacturers the iPhone, maintains the iOS operating system and operates the related App Store market. Such universal control is not the only model. Google strives for a similar comprehensive solution, maintaining the Android OS, operating the Google Play market, and working closely with manufacturers of "nexus" branded devices. However, Android is also readily available and is subsequently embraced by a host of other manufacturers. Such a model permits marketplace competition where manufacturers, cellular providers, enterprises, or nearly anyone may operate a compatible marketplace.

This dissertation supports the thesis that safe software dissemination among many marketplaces is possible. Compared to today, safety can be enhanced along many facets of a distributed dissemination model. In some cases, greater safety might be garnered by architectural changes to the software or hardware. In other cases, greater safety may hinge upon the actions of developers or consumers.

The ability to extend the base functionality of a smartphone is part of why the devices are so appealing to consumers. The applications that facilitate this extended functionality are created by developers, software creators that have varied familiarity with security-related topics. To aid developers in creating safer applications, we created a tool that integrates into common tools and workflows used to develop mobile applications. Our tool highlights least-privilege violations, encouraging developers to minimize the number of permissions requested by an application. In turn, this also reduces the burden of permission comprehension placed on end users.

Once deemed ready for dissemination, developers typically publish applications to one or more online marketplaces. For smartphones, these marketplaces are key components in software dissemination models, and users of these markets are fundamentally safer when market proprietors take an active role in policing their markets for malware. To facilitate this market safeguarding, we studied existing malware, particularly those employing evasive techniques. We extended techniques observed in contemporary malware to better understand the theoretical scope of such evasions—those likely to be embraced by future malware. Consequently, we developed a new analysis platform designed to both resist evasion and to better coerce malicious behavior than other modern analysis systems.

To provide safety in a more comprehensive way, we also introduced a new protocol, AppIntegrity, that provides developer-to-user software verification. So, orthogonal to a user's marketplace preference, the user can be assured that the software installed to their device is that which the developer intended. The assurance provided by AppIntegrity mitigates the risk presented by a technique employed by the vast majority of mobile malware, application repackaging. As implementations of AppIntegrity may require user-facing components, fundamental concerns of user deception and understanding are warranted. To better understand the scope and legitimacy of such concerns, we evaluated proposed AppIntegrity interfaces via several user studies. Our evaluation is promising, users appear to be able to take appropriate actions in response to AppIntegrity prompts—ultimately avoiding dangerous conditions.

While areas for future research clearly exist, we have provided opportunities for improved safety via architectural enhancements, developer aid, market proprietor aid, and more broadly with the AppIntegrity protocol. We hope that our ideas will improve the overall safety of software dissemination models and ultimately improve society.

7.1 Future Directions

We are clearly driven to provide safety to users of online software marketplaces. Toward this end, and beyond the future work already articulated in previous sections, we identify three directions for extending the research presented here: (1) AppIntegrity improvements toward adoption, (2) AppIntegrity Protocol enhancements, and (3) using AppIntegrity as a foundational technology.

Protocol improvements toward adoption. As mentioned in section 6.4.3, a primary detractor for wide-spread adoption of AppIntegrity is the approximately 25% negative response when applications successfully verify. Ideally, this rate approaches zero, however, in practice, such rates are often material. The decision to implement a security control may involve weighing a complicated trade-off. In the case of AppIntegrity, such decisions are at a minimum driven by complicated economic risk analysis and perhaps border upon generalized, societal safety. As an example of the potential economic complexity, consider the required inclusion for back-up cameras on automobiles in the U.S. by 2018. The National Highway Traffic Safety Administration (NHTSA) estimates the per-vehicle cost of adding such cameras to be \$44–\$140 USD. Taking into account the projected number of new vehicles and given that 59–69 of the 210 related deaths per year are expected to be saved, the NHTSA estimates the cost per life saved ranges from \$15.9–\$26.3 million USD [50]—well above the median overall \$42,000 (or even \$2.8 million for toxin control) cost effectiveness per life-year [215]. After careful consideration, the NHTSA has decided to make inclusion of such cameras mandatory, even with the considerable associated cost. We expect similar reasoning would need to occur in order to inform the deployment and tuning of a mechanism similar to AppIntegrity.

The interfaces evaluated in Chapter 6, are little more than what the rich field of Human-Computer Interaction (HCI) would consider "mockups." Considerable security HCI research exists [54, 67, 212], much of which may guide future refinement or complete re-design of our proposed UIs. Use of color, different design layouts, the precise verbiage presented to users and hosts of other factors have all already been studied under different contexts. Perhaps the most related is recent work to improve web browser safety [111]. Researchers were able to improve adherence rates by employing design cues that "promote the safe choice" and "demote the unsafe choice." Google's Chrome browser increase's safety by making it more difficult for a user to proceed to a site that may cause the user harm (Chrome similarly increases the burden placed on users to install potentially harmful SSL certificates). Over-arching guidelines for security-related UIs have yet to be established, but we are certain there is room to improve the user-facing components we presented in Chapter 6. Indeed, improving the interfaces may well improve the already-promising results established by our user studies.

In addition to improving UIs, AppIntegrity implementations could elect to display interfaces less frequently. For example, our user studies indicated that users perform well when AppIntegrity verification fails, representing high likelihood of attack (C-FAIL). If AppIntegrity were implemented in a manner were users were only notified when risk was high, (therefore not in the common case, when application verification succeeds) we might see enough continued market use for AppIntegrity to be acceptable to today's market proprietors. Such a convention is not completely alien to today's user. Consider web browser SSL operation, if the certificate has desirable properties, browsers typically inform the user via a small, unobtrusive notification (e.g. a small green icon in the URL bar). However, if the certificate has undesirable properties, the user may see a large, red "X" or even a model / interstitial dialog hindering the user from visiting the likely harmful site. However, implementers should be aware that implementing a security-related interface such that it seems to appear randomly to users, may give the appearance (accurate or not) that the control is not following the principle of complete mediation [198]. Users that are conditioned to enter a password only at expected times in a sequence (e.g. logging into the computer following boot or a screen lock) are less susceptible to providing a password to malware that simply prompts the user for a password.

Building upon AppIntegrity Instead of only displaying UIs based on AppIntegrity conditions, many forms of higher-level technology could be employed to influence when prompts are displayed or as a component in a future prompt design. For instance, there has been considerable success in "crowdsourcing" or "wisdom of crowds" approaches to security [130, 142, 245]. By collecting AppIntegrity UI responses, one might envision only displaying prompts to randomly sample users until an application-domain pair falls below a threshold. Similarly, a modified design of an AppIntegrity prompt might have a market-like component that informs users that "83% of users decided to install the application following this prompt" (a technique that has shown success in Phishing research [245]).

Many variations of UI modification could take advantage of device-local or broad (centralized or decentralized) databases regarding the legitimacy of certificates and the application-domain pairs. Systems like Perspectives [237] and other higher-level protocols discussed in section 6.6 can leverage AppIntegrity's foundation to provide safety to users in new ways. Further, as with the crowdsourcing example above, higher-layer protocols could similarly improve the effectiveness of AppIntegrity. Such improvement may prove to be the best way to decrease negative responses in reaction to correct verification.

Protocol Enhancements The process of "benevolent repackaging," introduced in section 6.7, would be deterred by AppIntegrity adoption. AppIntegrity ensures that software a user installs is that which a developer intended – third party repackaging is hindered regardless of malicious of benevolent intent. The protocol presented in section 6.2 might be altered to support certain types of repackaging or general repackaging by vetted entities. Of course, complex PKI components could be leveraged in addition to the relatively simple PKCS components of AppIntegrity. However, there may also be protocol enhancements that facilitate third-party support without the need for a PKI or other systems that detract from the simplicity and distributed nature of AppIntegrity. For instance, perhaps multiple keys could be held in the `android.cert` file, each allowing for verification of application components, but with independent revocation options. Original developers might issue secondary keys for specific purposes, thus allowing all components to verify in some way. Models might include signing of keys provided by third parties (which may introduce other identity issues), or other more complicated PKCS features. The complexity of this direction warrants substantial consideration, but may be required in the future as mobile application development processes become increasingly complex.

Chapter 8

Appendices

8.1 A5 Malware Plugin Example

```
#filenames for plugins must start with the string "plugin_" and end in ".py"

#plugins always return a tuple (pluginName,listOfCountermeasures,listOfComments)
#where the first value is a string and the second two are each a python List
#plugins should have no printed output

#pluginName is a required variable for plugins
#this is simply a name for the plugin that is used in logging and stdout

pluginName = "NotCompatible data decyptor"

#if true, the plugin will be used, if false it will not
enable = True

#type is a required variable for plugins
#type is simply a string that is used to group plugins by category,
#used for unit testing, in production the type often doesn’t matter
type = "known"

#logger is optional, if the plugin requests a logger like this, logging entries will end up in the shared log
#import logging
#logger = logging.getLogger(__name__)

class PluginClass:
    msg = "autocreated NotCompatible data decryptor"
    def get_dns_rule(self,domain):
      return ('alert udp any any -> any 53 (msg:"%s"; content:"%s"; nocase;)'
             % (self.msg,domain))

    def get_notc_rule(self,port):
      return ('alert tcp any any -> any %s (msg:"%s"; flow:established,
             to_server; dsize:13; content:"|04|"; depth:1;
             content:"|01 05 00 00 00 00 07 00|";)' % (port,self.msg))

    def run(self,pcap,apk):
        ruleList = list()
        commentList = list()

        if pcap is None or apk is None:
          commentList.append("this plugin requires a pcap file and an apk to work")
          logger.error("plugin requires a pcap and an apk...but didn't get em")
          return (pluginName,None,commentList)

        try:
          from scapy.all import PcapReader,hexdump,ls
          import sys

          my_reader = PcapReader(pcap)
          if(self.findNotCompatiblePhoneHome(my_reader)):
            pt = self.decryptNotCompatibleData(apk,ruleList,commentList)
            (primary,secondary,pport,sport) = pt.split('|')
            commentList.append("new notcompatible sockets: %s:%s , %s:%s" % (primary,pport,secondary,sport))
            ruleList.append(self.get_dns_rule(primary))
            ruleList.append(self.get_dns_rule(secondary))
            ruleList.append(self.get_notc_rule(pport))
            ruleList.append(self.get_notc_rule(sport))
        except IOError:
          logger.error("Failed reading pcap")
          return (pluginName, None, None)

        return (pluginName, ruleList, commentList)
    def decrypt(self,passkey,param):
        from Crypto.Hash import SHA256
        from Crypto.Cipher import AES
        keyhash = SHA256.new(passkey)
        key = keyhash.digest()
        cipher = AES.new(str(key), AES.MODE_ECB, "ignored")

        buf = buffer(param,0,len(param))
        return cipher.decrypt(buf)

    def decryptNotCompatibleData(self,apk,rules,comments):
        from scapy.all import hexdump

        #directly from the zip (relies upon working zip imlementation in python!)
        import zipfile
        file = zipfile.ZipFile(apk,"r")
        data = file.read("res/raw/data")

        key = "ZTY4MGE5YQo"
        comments.append("key: %s" % key)
        plaintext = self.decrypt(key, data)
        comments.append("decrypted: %s" % plaintext)

        return plaintext

    def findNotCompatiblePhoneHome(self,reader):
      from scapy.all import TCP,Raw,hexdump
      for pkt in reader:
        if pkt.haslayer(TCP):
          if pkt.haslayer(Raw):
            data = pkt.getlayer(Raw).load
            if len(data) == 13:
              if data[0] == '\x04':
                initialcall = bytearray.fromhex("01 05 00 00 00")
                if data[3:8] == initialcall:
                  return True
      return False
```

8.2 Using Reflection to Obtain System Properties

```java
private void logRuntimeSystemPropsReflect(){
  logAprop("ro.secure");
  logAprop("ro.product.name");
  logAprop("ro.debuggable");
  logAprop("status.battery.level_raw");
  logAprop("ro.build.host");
  logAprop("ro.build.tags");
  logAprop("net.gprs.local-ip");
  logAprop("net.eth0.gw");
  logAprop("net.dns1");
  logAprop("gsm.operator.numeric");
  logAprop("ro.kernel.qemu");
  logAprop("ro.kernel.qemu.gles");
  logAprop("ro.kernel.android.qemud");
}
private void logAprop(String s){
  Log.e("reflect."+s,getPropViaReflect(s));
}

//the actual reflection to obtain a handle to the hidden android.os.SystemProperties
private String getPropViaReflect(String s){
  String ret="";
  try{
    ClassLoader cl = theActivity.getBaseContext().getClassLoader();
    Class<?> SystemProperties = cl.loadClass("android.os.SystemProperties");

    @SuppressWarnings("rawtypes")
    Class[] paramTypes = { String.class };
    Method get = SystemProperties.getMethod("get", paramTypes);

    Object[] params = { s };
    ret = (String) get.invoke(SystemProperties, params);
  } catch(Exception e){
    e.printStackTrace();
  }
  return ret;
}
```

Figure 8.1: **Obtaining system properties using reflection.** This listing uses reflection to obtain runtime `SystemProperties` information. The information is logged to the system log, for detection purposes this value would be evaluated in accordance with the detection techniques presented section 5.5. This code listing obtains information without using the official API and without requiring additional permissions. This code obtains values for the battery level, the `Build` configuration, network IP settings, and cellular provider MCC and MNC.

```java

private void logRuntimeSystemPropsExec(){
  try
  {
    String line;
    java.lang.Process p = Runtime.getRuntime().exec("getprop");
    BufferedReader input = new BufferedReader(new
        InputStreamReader(p.getInputStream()));
    while ((line = input.readLine()) != null)
    {
      //quick line parsing
      int split = line.indexOf("]: [");
      String k = line.substring(1,split);
      String v = line.substring(split+4,line.length()-1);
      Log.e("runprop."+k,v);
    }
    input.close();
  }
  catch (Exception err)
  {
    err.printStackTrace();
  }
}
```

Figure 8.2: **Obtaining system properties using runtime exec.** This listing uses "runtime exec" to obtain runtime `SystemProperties` information utilizing the `getprop` command. The in formation is logged to the system log, for detection purposes this value would be evaluated in accordance with the detection techniques presented section 5.5. The code listed here obtains `SystemProperties` information without using the official API and without requiring additional permissions.

8.3 Google Consumer Surveys Overview

User studies detailed in section 6.4 were conducted using Google Consumer Surveys (GCS), an online market survey tool. Researchers may create and pose up to ten questions to each participants at a rate of \$0.10 per question or \$1.10–\$3.50 per survey. GCS guarantees a minimum number of responses, that is, if a researcher requests (and pays for) 100 participants, GCS will deliver at least 100 results. The URL for GCS is `https://www.google.com/insights/consumersurveys/home`.

GCS operates in two capacities, a web-based survey platform, and an Android-based survey platform. Our studies exclusively used the Android survey mechanism, meaning all of our participants were Android users, and the survey was actually taken using an Android device. As GCS is a Google product, Google supports the service (e.g. helpdesk, technical support). Participant identities remain anonymous to researchers, and Google screens surveys prior to launch to protect users (e.g. anonymity violation, offensive language, unlawful purposes).[1]

Google Consumer Surveys has been shown to produce highly accurate results, both in comparison to Probability and Non-Probability based Internet surveys [175] and traditional surveys [200]. In most cases GCS outperforms Internet-based surveys (with respect to a benchmark) and is within three percentage points compared to dual-frame telephone surveys.

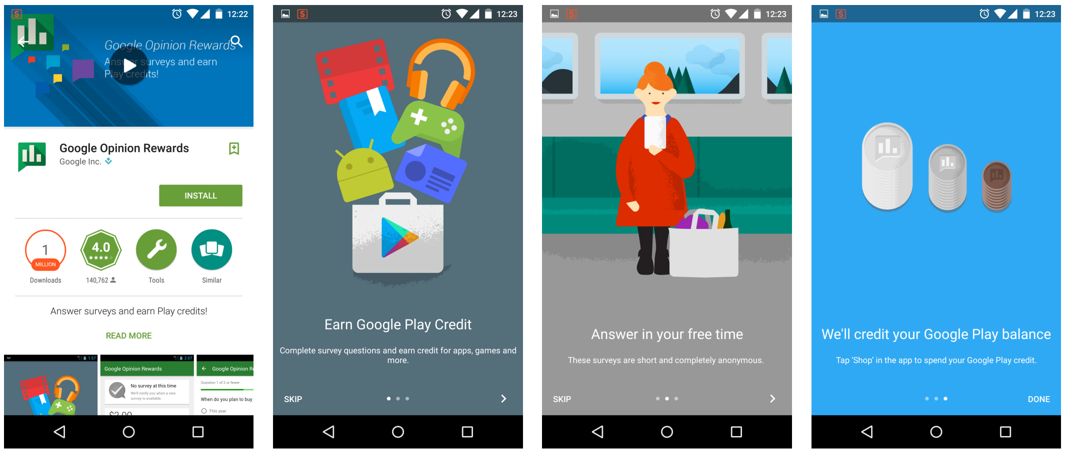
Unlike traditional user study recruitment, GCS users opt-in to participation using the "Google Opinion Rewards" application, found in Google Play. Android users may happen upon the application, Google may promote the application using standard Google Play application promotion techniques, users may read about "Google Opinion Rewards" in the press, etc. As of Jan 2016, the application has been installed at least five million times and has received 177,047 5-star reviews (of 308,877 total) netting an overall 4.0 rating (of 5) on Google Play.

When a user first launches the application, they are greeted with several "splash screens" offering high-level instructions on use (depicted in Figure 8.3). The user is then immediately prompted with a survey, proctored using the standard GCS interface. The purpose of the initial survey is twofold: first to obtain participant consent (Figure 8.4), and to collect demographics (Figure 8.5). Unlike subsequent surveys, the first survey is compulsory.

Unlike other survey mechanisms, researchers need not pose questions relating to any of the demographics listed in Figure 8.5. The urban density and location questions must be answered. Other demographic questions allow for free-text input (e.g. gender, occupation), or omission (e.g. age, income). Demographics are collected (and reported) as pre-defined, constant groupings. The groups can also be observed in Figure 8.5. For instance, age for each participant is recorded as one of: 18–24, 25–34, 35–44, 45–54, 55–64, and 65 or older.

While anonymous response demographics are available to the researcher, for surveys with more than 250 participants, GCS also reports deviation from the surveyed responses from United States Census Bureau [113]. The deviation

[1]The policies governing the product can be found at `https://support.google.com/consumersurveys/answer/2375134`



Answer surveys and earn Play credits!

Earn Google Play Credit

Complete survey questions and earn credit for apps, games and more.

Answer in your free time

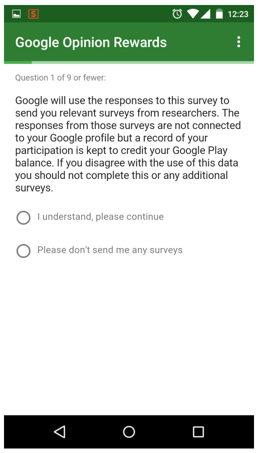
These surveys are short and completely anonymous.

We'll credit your Google Play balance

Tap 'Shop' in the app to spend your Google Play credit.

(a) Google Play install screen (b) Splash screen A (c) Splash screen B (d) Splash screen C

Figure 8.3: **GCS sign-up process**: Screens presented in GCS sign-up. Figure 8.3(a) is presented from Google Play prior to installation. Figures 8.3(b)–8.3(d) are presented upon initial launch of the application.



Question 1 of 9 or fewer:

Google will use the responses to this survey to send you relevant surveys from researchers. The responses from those surveys are not connected to your Google profile but a record of your participation is kept to credit your Google Play balance. If you disagree with the use of this data you should not complete this or any additional surveys.

○ I understand, please continue

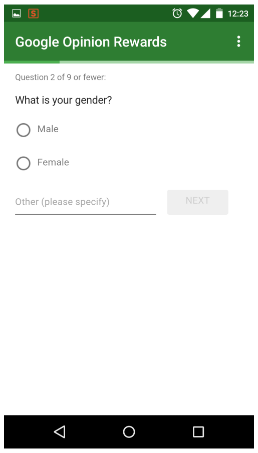
○ Please don't send me any surveys

Figure 8.4: **GCS consent screen**: Upon initial launch of the application, a short survey is immediately presented. The first question of this automatic survey explains that a record of survey participation is kept in conjunction with the participant's Google account, which is also how the participant is compensated.

is calculated as a RMSE, a "weighted average of the difference between the predicted population sample and the actual sample." Smaller values indicate lower bias in the sample (RMSE data can be found in section 6.4 and in greater detail in appendix 8.4).

Users that install the application may later (possibly days or weeks) be notified that a new survey is available, using an Android notification bar icon as shown in Figure 8.6. The user may ignore this notification, or explicitly partake or opt-out of the related survey. In this way, users may decline to participate in any survey.

Users are compensated "up to $1.00" (USD) in Google Play credit, upon completing each survey. No guarantees are made by Google regarding the distribution of compensation between 0 and $1.


Google Opinion Rewards

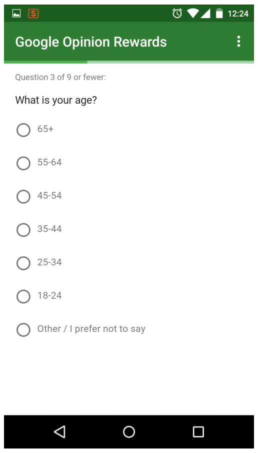
Question 2 of 9 or fewer:

What is your gender?

- Male
- Female

Other (please specify) NEXT

(a) Gender

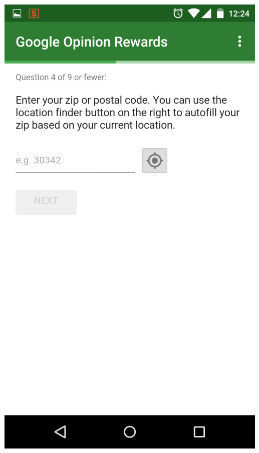

Google Opinion Rewards

Question 3 of 9 or fewer:

What is your age?

- 65+
- 55-64
- 45-54
- 35-44
- 25-34
- 18-24
- Other / I prefer not to say

(b) Age


Google Opinion Rewards

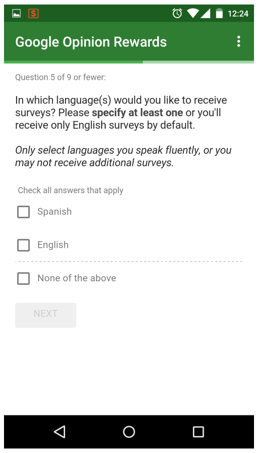
Question 4 of 9 or fewer:

Enter your zip or postal code. You can use the location finder button on the right to autofill your zip based on your current location.

e.g. 30342

NEXT

(c) Location


Google Opinion Rewards

Question 5 of 9 or fewer:

In which language(s) would you like to receive surveys? Please **specify at least one** or you'll receive only English surveys by default.

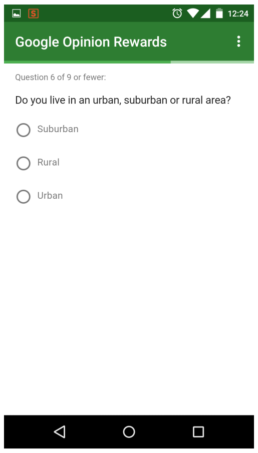
Only select languages you speak fluently, or you may not receive additional surveys.

Check all answers that apply

- Spanish
- English
- None of the above

NEXT

(d) Language

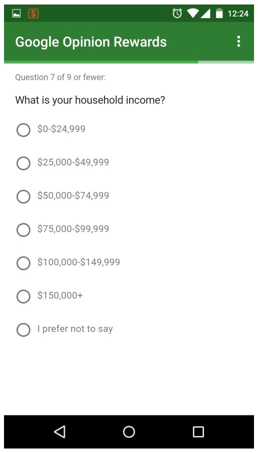

Google Opinion Rewards

Question 6 of 9 or fewer:

Do you live in an urban, suburban or rural area?

- Suburban
- Rural
- Urban

(e) Urban density

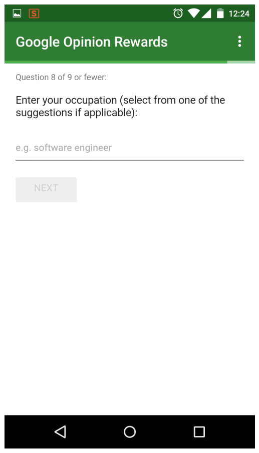

Google Opinion Rewards

Question 7 of 9 or fewer:

What is your household income?

- $0-$24,999
- $25,000-$49,999
- $50,000-$74,999
- $75,000-$99,999
- $100,000-$149,999
- $150,000+
- I prefer not to say

(f) Household income


Google Opinion Rewards

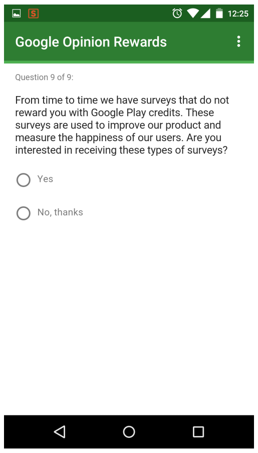
Question 8 of 9 or fewer:

Enter your occupation (select from one of the suggestions if applicable):

e.g. software engineer

NEXT

(g) Occupation


Google Opinion Rewards

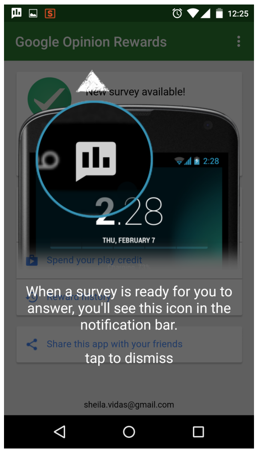
Question 9 of 9:

From time to time we have surveys that do not reward you with Google Play credits. These surveys are used to improve our product and measure the happiness of our users. Are you interested in receiving these types of surveys?

- Yes
- No, thanks

(h) Additional research opt-in

Figure 8.5: **GCS demographic collection**: Upon initial launch of the application, a short survey is immediately presented. The second and following questions of this automatic survey collect demographic information about the participant. These demographics are not collected with every following survey.



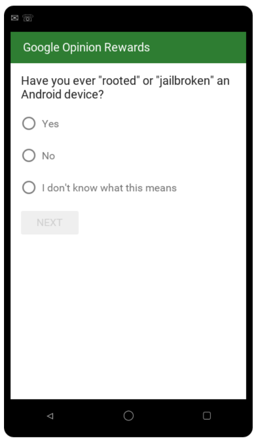
When a survey is ready for you to answer, you'll see this icon in the notification bar.

tap to dismiss

Figure 8.6: **GCS sign-up completion**: Following completion of the initial survey, the user is informed that an icon will later appear in the Android notification bar when a survey is available.

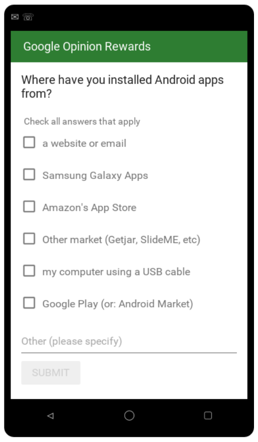
Surveys are advertised to have responses appear “immediately” with full completion expected withing seven days. In our experience, surveys only took a few days to complete.

As of Jan 2016, GCS is available in United States, Canada, United Kingdom, Mexico, Japan, Australia, Brazil, Spain, France, and Germany.


Google Opinion Rewards

Have you ever "rooted" or "jailbroken" an Android device?

(a) C-ROOT

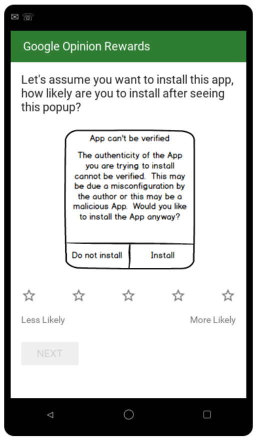

Google Opinion Rewards

Where have you installed Android apps from?

Check all answers that apply

a website or email
Samsung Galaxy Apps
Amazon's App Store
Other market (Getjar, SlideME, etc)
my computer using a USB cable
Google Play (or: Android Market)
Other (please specify)
SUBMIT

(b) C-MARKET

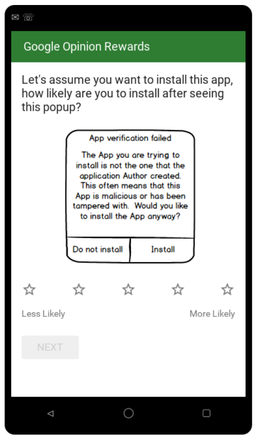

Google Opinion Rewards

Let's assume you want to install this app, how likely are you to install after seeing this popup?

App can't be verified

The authenticity of the App you are trying to install cannot be verified. This may be due a misconfiguration by the author or this may be a malicious App. Would you like to install the App anyway?

(c) C-CANTV

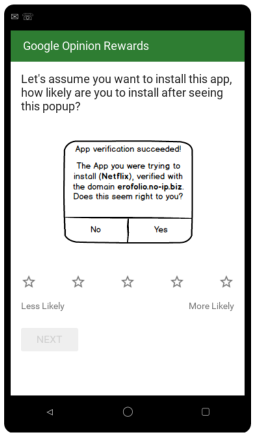

Google Opinion Rewards

Let's assume you want to install this app, how likely are you to install after seeing this popup?

App verification failed

The App you are trying to install is not the one that the application Author created. This often means that this App is malicious or has been tampered with. Would you like to install the App anyway?

(d) C-FAIL

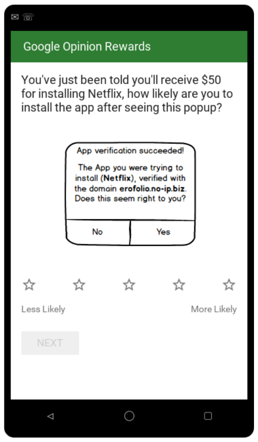

Google Opinion Rewards

Let's assume you want to install this app, how likely are you to install after seeing this popup?

App verification succeeded!

The App you were trying to install (**Netflix**), verified with the domain **erofolio.no-ip.biz**. Does this seem right to you?

(e) C-PASSFP

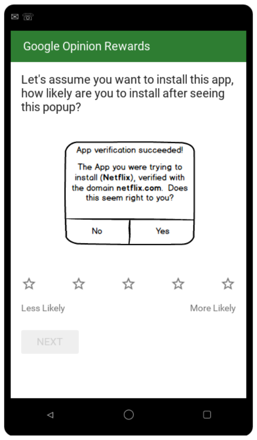

Google Opinion Rewards

You've just been told you'll receive $50 for installing Netflix, how likely are you to install the app after seeing this popup?

App verification succeeded!

The App you were trying to install (**Netflix**), verified with the domain **erofolio.no-ip.biz**. Does this seem right to you?

(f) C-PASSFP50


Google Opinion Rewards

Let's assume you want to install this app, how likely are you to install after seeing this popup?

App verification succeeded!

The App you were trying to install (**Netflix**), verified with the domain **netflix.com**. Does this seem right to you?

(g) C-PASS

Figure 8.7: **GCS AppIntegrity interface study depictions**: User interface depiction for each condition in the AppIntegrity Interface study discussed in section 6.4.

8.4 User Study Materials

User studies detailed in section 6.4 were conducted using the Android implementation of Google Consumer Surveys (GCS) (see appendix 8.3). We further limited our studies to only include U.S. users 18 and older, using a participant pool limitation capability native to GCS.

Each of the AppIntegrity Interface study conditions from section 6.4 are depicted in Figure 8.7.

Bibliography

[1] AMAT: Android Malware Analysis Toolkit. http://sourceforge.net/projects/amatlinux/. 95, 109

[2] Androguard: Reverse engineering, malware and goodware analysis of android applications. http://code.google.com/p/androguard/. 77, 92

[3] Android developer documentation. http://developer.android.com/. 10, 11, 12, 13, 14, 16

[4] Andrubis. http://anubis.iseclab.org/. 72, 90, 95, 109

[5] Applications using webkit - webkit. http://trac.webkit.org/wiki/Applications%20using%20WebKit. 15

[6] CopperDroid. http://copperdroid.isg.rhul.ac.uk/copperdroid/. 95, 109

[7] Droidbox device identifier patch. https://code.google.com/p/droidbox/source/browse/trunk/droidbox23/framework_base.patch?r=82. 113

[8] DuckDuckGo mobile application source code. https://github.com/duckduckgo/android. 66

[9] Easy root. http://www.unstableapps.com/buyme.html. 20

[10] Factsheet: The U.S. media universe | nielsen wire. http://blog.nielsen.com/nielsenwire/online_mobile/factsheet-the-u-s-media-universe/. 8

[11] FootPath source code. https://github.com/COMSYS/FootPath. 66

[12] Foresafe. http://www.foresafe.com/scan. 109

[13] Google Authenticator source code. https://github.com/google/google-authenticator-android. 66

[14] Hockey Android source code. https://github.com/bitstadium/HockeyAndroid. 66

[15] Imperial Calculator source code. https://github.com/nolanlawson/imperial-calculator-android. 66

[16] Lookout mobile security - android market. https://market.android.com/details?id=com.lookout. 17

[17] mobile-sandbox. http://mobilesandbox.org/. 110

[18] Monitoring the Battery Level and Charging State | Android Developers. http://developer.android.com/training/monitoring-device-state/battery-monitoring.html. 106

[19] Normal permissions. http://developer.android.com/guide/topics/security/normal-permissions.html. 11

[20] North American Numbering Plan Adminstration search. www.nanpa.com/enas/area_code_query.do. 97

[21] Open handset alliance. http://www.openhandsetalliance.com/. 13

[22] Permission list is incorrect for apks built with android 1.6 sdk. https://code.google.com/p/android/issues/detail?id=4101. 10

[23] Picasso source code. https://github.com/Pixate/picasso. 66

[24] Platform versions - current distribution. http://developer.android.com/resources/dashboard/platform-versions.html. 14

[25] Root tools. https://market.android.com/details?id=com.jrummy.roottools. 20

[26] SandDroid. http://sanddroid.xjtu.edu.cn/. 95, 109

[27] Soar Android source code. https://github.com/SoarGroup/Domains-RoomsWorld. 66

[28] Track Map source code. https://github.com/lociii/TrackMap. 66

[29] Trojanized apps root android devices. http://blog.trendmicro.com/trojanized-apps-root-android-devices/. 19

[30] Unix system family tree: Research and bsd. http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/share/misc/bsd-family-tree?rev=HEAD. 15

[31] Using the Android Emulator | Android Developers. http://developer.android.com/tools/devices/emulator.html. 102

[32] What is android. http://developer.android.com/guide/basics/what-is-android.html. 8

[33] Yaxim Android source code. https://github.com/pfleidi/yaxim. 66

[34] Zeus targets mobile users. http://blog.trendmicro.com/zeus-targets-mobile-users/. 19

[35] Just because it's signed doesn't mean it isn't spying on you. http://www.f-secure.com/weblog/archives/00001190.html, May 2007. 18

[36] Worm:symbos/yxe. http://www.f-secure.com/v-descs/worm_symbos_yxe.shtml, May 2007. 19

[37] Android most popular operating system in us among recent smartphone buyers|nielsen wire. http://blog.nielsen.com/nielsenwire/online_mobile/android-most-popular-operating-system-in-u-s-among-recent-smartphone-buyers/, Oct. 2010. 8

[38] Android.com. http://www.android.com/privacy.html, Oct. 2010. 60

[39] Posit-mobile issue 100. http://code.google.com/p/posit-mobile/issues/detail?id=100, Nov. 2010. 65

[40] Rooting the droid without rsdlite. http://androidforums.com/droid-all-things-root/171056-rooting-droid-without-rsd-lite-up-including-frg83d.html, Dec. 2010. 21

[41] selenium revision log. http://code.google.com/p/selenium/source/diff?path=/trunk/android/server/AndroidManifest.xml&format=side&r=10729&old_path=/trunk/android/server/AndroidManifest.xml&old=10639, Dec. 2010. 65

[42] Amazon appstore frequently asked questions. https://developer.amazon.com/help/faq.html, Oct. 2011. 126

[43] android-apktool: a tool to reverse engineer Android apk files, 2011. https://code.google.com/p/android-apktool/. 76, 119

[44] Android developer guide 4.0 r1. http://developer.android.com/reference/, Oct. 2011. 83, 84

[45] Android developer guide 4.0 r1. http://developer.android.com/guide/topics/manifest/manifest-element.html, Oct. 2011. 118

[46] android4me: J2ME port of Google's Android, 2011. https://code.google.com/p/android4me/downloads/list. 76, 119

[47] Canvas: Owning android. http://partners.immunityinc.com/movies/Lightning_Demo_Android.zip, Jan. 2011. 15, 20

[48] Security alert: Malware found targeting custom roms. http://blog.mylookout.com/2011/06/security-alert-malware-found-targeting-custom-roms-jsmshider/, June 2011. 126

[49] "Update to the AutoPlay functionality in Windows, KB971029". Oct 2012. 48

[50] Federal motor vehicle safety standards; rear visibility. *National Highway Traffic Safety Administration*, pages 19177–19250, Apr. 2014. 151

[51] A. Acquisti and J. Grossklags. Privacy and rationality in individual decision making. *Security & Privacy, IEEE*, 3(1):26–33, 2005. 58

[52] R. Adame. Avg community powered threat report - q1 2011, Apr. 2011. 43

[53] Y. Agarwal, M. Hall, P. Gupta, L. Dolecek, N. Dutt, R. Gupta, R. Kumar, S. Mitra, A. Nilolau, T. Rosing, et al. Protectmyprivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. *Currently Under Review*, 2013. 145

[54] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Usenix security*, pages 257–272, 2013. 151

[55] R. Anderson and T. Moore. The economics of information security. *Science*, 314(5799):610–613, 2006. 28

[56] I. Asrar. Could sexy space be the birth of the sms botnet? http://www.symantec.com/connect/blogs/could-sexy-space-be-birth-sms-botnet, July 2009. 117

[57] I. Asrar. A touch of mobile threat dÃľjÃă vu. http://www.symantec.com/connect/blogs/touch-mobile-threat-deja-vu, Feb. 2010. 117

[58] I. Asrar. Will sms bring you free vouchers? http://www.symantec.com/connect/blogs/will-sms-bring-you-free-vouchers, Apr. 2010. 47, 117

[59] I. Asrar. Will your next tv manual ask you to run a scan instead of adjusting the antenna?, Oct. 2011. 119

[60] I. Asrar, A. Hinchliffe, B. Kay, and A. Verma. Who's watching you?: Mcafee mobile security report. http://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf, Feb. 2014. 44

[61] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010. 9, 24, 65

[62] A. Barth, A. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*. Citeseer, 2010. 67

[63] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*. Citeseer, 2009. 72, 84, 95

[64] M. Becher, F. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proc. IEEE Symp. on Security and Privacy*, pages 96–111. IEEE, 2011. 116, 124

[65] Y. Benjamin. Apple privacy score - snow leopard - 10, iphone - 0. `http://blog.sfgate.com/ybenjamin/2009/08/27/apple-privacy-score-snow-leopard-10-iphone-0/`, Aug. 2009. 43

[66] T. Blasing, L. Batyuk, A. Schmidt, S. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *MALWARE'10*, 2010. 72, 95

[67] C. Bravo-Lillo, L. Cranor, S. Komanduri, S. Schechter, and M. Sleeper. Harder to ignore? revisiting pop-up fatigue and approaches to prevent it. In *10th Symposium On Usable Privacy and Security (SOUPS 2014)*, pages 105–111, 2014. 151

[68] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symp. S.& P. 2008*, pages 143–157, May 2008. 14

[69] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proc. ACM work. Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011. 55, 143

[70] M. C. Don't stand so close to me: An analysis of the nfc attack surface. july 2012. 35

[71] J. V. Camp. Only 36.2 percent of android devices run froyo. `http://www.digitaltrends.com/mobile/only-36-2-percent-of-android-devices-run-froyo/`, Nov. 2010. 13

[72] T. Cannon. Android lock screen bypass. `http://thomascannon.net/blog/2011/02/android-lock-screen-bypass`. 19

[73] C. Castillo. Android malware past, present, and future. Nov. 2011. 32, 43

[74] D. J. Chaboya, R. A. Raines, R. O. Baldwin, and B. E. Mullins. Network intrusion detection: automated and manual methods prone to attack and evasion. *Security & Privacy, IEEE*, 4(6):36–43, 2006. 95

[75] A. Chaudhuri. Language-based security on android. In *ACM SIGPLAN Workshop on Prog. Lang. and Analysis for Security*, pages 1–7, 2009. 59, 67

[76] E. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proc. ACM CCS*, pages 227–238. ACM, 2011. 143

[77] J. Chen. Aurora feint iphone app delisted for lousy security practices. `http://gizmodo.com/5028459/aurora-feint-iphone-app-delisted-for-lousy-security-practices`, 2008. 43

[78] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. IEEE International Conference on*, pages 177–186, 2008. 95

[79] X. Chen, T. Dirro, P. Greve, H. Li, F. Paget, C. Schmugar, J. Shah, R. Sherstobitoff, D. Sommer, B. Sun, and A. Wosotowsky. Mcafee threats report: First quarter 2013. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2013.pdf`, July 2013. 44

[80] J. Cheng, S. Wong, H. Yang, and S. Lu. Smartsiren: virus detection and alert for smartphones. In *Proc. ACM MobiSys*, pages 258–271, 2007. 125

[81] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *9th Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2011. 73

[82] N. Christin, A. Weigend, and J. Chuang. Content availability, pollution and poisoning in file sharing peer-to-peer networks. In *Proceedings of the 6th ACM conference on Electronic commerce*, pages 68–77. ACM, 2005. 119

[83] F. Chytry. How are you doing mr. android? `https://blog.avast.com/2014/01/29/how-are-you-doing-mr-android/`, Jan. 2014. 43

[84] D. Cooper. Lg commits to monthly android security updates. Aug. 2015. 23

[85] M. Cooper, R. W. Dronsuth, A. J. Mikulski, C. N. Lynk Jr, J. J. Mikulski, J. F. Mitchell, R. A. Richardson, and J. H. Sangster. Radio telephone system, Sept. 16 1975. US Patent 3,906,166. 31

[86] H. Daharmdasani. Android.hehe: Malware now disconnects phone calls. http://www.fireeye.com/blog/technical/2014/01/android-hehe-malware-now-disconnects-phone-calls.html, Jan. 2014. 81

[87] C. Davies. Sprint htc hero android 2.1 update released. `http://www.slashgear.com/sprint-htc-hero-android-2-1-update-released-1986138/`, May 2010. 14

[88] R. Dawkins and J. R. Krebs. Arms races between and within species. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 205(1161):489–511, 1979. 29, 42, 45

[89] A. K. Dewdney. In the game called core war hostile programs engage in a battle of bits. In *Scientific American*, May 1984. 28, 30

[90] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590. ACM, 2006. 138

[91] DHS. Roll call release: Threats to mobile devices using the android operating system. `http://info.publicintelligence.net/DHS-FBI-AndroidThreats.pdf`, 2013. 44

[92] J. S. Downs, M. B. Holbrook, and L. F. Cranor. Decision strategies and susceptibility to phishing. In *Proceedings of the second symposium on Usable privacy and security*, pages 79–90. ACM, 2006. 138

[93] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. CCS*, pages 2–11. ACM, 2004. 95

[94] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011. 144

[95] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 326–343. IEEE, 1989. 29, 30

[96] P. Elmer-DeWitt. Who's minding the apple app store? `http://fortune.com/2015/11/13/apple-app-store-error/`, Nov. 2015. 4

[97] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: an Information-Flow tracking system for realtime privacy monitoring on smartphones. In *OSDI 2010*, Vancouver, BC, Canada. 18, 24, 67, 73, 144

[98] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proc. of the 20th USENIX Security Symposium*, 2011. 78, 89, 119

[99] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. ACM CCS*, pages 235–245, Chicago, IL. 10, 19, 24, 144

[100] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security and Privacy*, Jan 2009. 76

[101] R. Ensafi, M. Jacobi, and J. R. Crandall. A case study in helping students to covertly eat their classmates. *Summit on Gaming, Games, and Gamification in Security Education*, Aug 2014. 30

[102] R. Entner. International comparisons: The handset replacement cycle. June 2011. 33

[103] R. Entner. 2014 us mobile phone sales fall by 15% and handset replacement cycle lengthens to historic high. Feb. 2015. 33

[104] M. F. and P. Schulz. Detecting android sandboxes, Aug 2012. `https://www.dexlabs.org/blog/btdetect`. 99

[105] F-secure. Mobile threat report q4 2012. `https://www.f-secure.com/documents/996508/1030743/Mobile+Threat+Report+Q4+2012.pdf`, mar 2013. 44

[106] F-Secure Labs. Mobile Threat Report Q3 2012. Technical report, Nov. 2012. 72

[107] D. Farmer and W. Venema. *Forensic discovery*. Addison-Wesley, 2005. 72, 93

[108] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011. 68, 77

[109] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. ACM work. Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011. 40, 70, 72, 143

[110] A. Felt and D. Wagner. Phishing on mobile devices. In *IEEE Workshop on Web 2.0 Security and Privacy*, 2011. 143

[111] A. P. Felt, A. Ainslie, R. W. Reeder, S. Consolvo, S. Thyagaraja, A. Bettes, H. Harris, and J. Grimes. Improving ssl warnings: comprehension and adherence. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2893–2902. ACM, 2015. 151

[112] P. Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007. 94

[113] T. File and C. Ryan. Computer and internet use in the united states: 2013. *American Community Survey Reports*, 2014. 131, 158

[114] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proc. CCS*, pages 59–68. ACM, 2006. 95

[115] Fortinet. 2014 threat landscape report. `http://www.fortinet.com/sites/default/files/whitepapers/Threat-Landscape-2014.pdf`, Feb. 2014. 44

[116] Gartner. Gartner says worldwide sales of mobile phones declined 3 percent in third quarter of 2012; smartphone sales increased 47 percent. `http://www.gartner.com/it/page.jsp?id=2237315`, Nov. 2012. 70

[117] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. Pfleeger, J. Quarterman, and B. Schneier. Cyberinsecurity: The cost of monopoly. *Computer and Communications Industry Association*, 2003. 8

[118] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM, 2011. 93

[119] A. Gostev. Kaspersky security bulletin 2007: Malware evolution in 2007. `https://securelist.com/analysis/kaspersky-security-bulletin/36192/kaspersky-security-bulletin-2007-malware-evolution-in-2007/`, Feb. 2008. 43, 44

[120] B. A. Graham. Improvement in telegraphy, Mar. 7 1876. US Patent 174,465. 31

[121] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security*, 2001. 95

[122] E. Hardy. Andoird OS 2.1 upgrade for T-Mobile myTouch 3G, Motorola Cliq, Cliq XT coming in August. `http://www.brighthand.com/default.asp?newsID=16767`, July 2010. 14

[123] A. Heath. Apple's tim cook declares the end of the pc and hints at new medical product. *The Telegraph*, Nov. 2015. 2

[124] M. Helft. Meet cyanogen, the startup that wants to steal android from google. mar 2015. 132

[125] J. Herrman. Android 2.1 update finally live for Motorola Droid. `http://gizmodo.com/#!5505520/android-21-update-finally-live-for-motorola-droid`, Mar. 2010. 14

[126] K. J. Higgins. Researcher to release smartphone botnet proof-of-concept code. `http://mobile.darkreading.com/9287/show/54cc009853e3f863244c0b267a73ae86`, Jan. 2011. 10

[127] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 29–36. IEEE, 2005. 95

[128] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proc. ACM CCS*, pages 639–652, Chicago, IL, 2011. ACM. 144

[129] M. Hypponen. State of cell phone malware in 2007. `https://www.usenix.org/legacy/event/sec07/tech/hypponen.pdf`, Aug. 2007. 44

[130] Q. Ismail, T. Ahmed, A. Kapadia, and M. K. Reiter. Crowdsourced exploration of security configurations. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 467–476. ACM, 2015. 152

[131] jgor. Android 5.x lockscreen bypass (cve-2015-3860). `http://sites.utexas.edu/iso/2015/09/15/android-5-lockscreen-bypass/`, sept 2015. 23

[132] J. John, F. Yu, Y. Xie, M. Abadi, and A. Krishnamurthy. deSEO: Combating search-result poisoning. In *Proceedings of USENIX Security 2011*, San Francisco, CA, Aug. 2011. 119

[133] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying spamming botnets using botlab. In *NSDI*, volume 9, pages 291–306, 2009. 28

[134] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In *USEC'12*, pages 68–79. Springer, 2012. 109, 124

[135] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286, 2004. 73

[136] C. Kimmel. Threat update: Rise in mobile malware. `http://blog.securestate.com/threat-update-rise-in-mobile-malware/mobile-malware-growth-chart/`, Oct. 2012. 43

[137] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1):51–56, 2004. 73

[138] P. Lantz, A. Desnos, and K. Yang. DroidBox: Android application sandbox. `http://code.google.com/p/droidbox/`, 2012. 73, 95, 113

[139] B. Lau and V. Svajcer. Measuring virtual machine detection in malware using dsd tracer. *Journal in Computer Virology*, 6(3):181–195, 2010. 94, 95

[140] B. Leiba and J. Fenton. Domainkeys identified mail (dkim): Using digital signatures for domain verification. In *Proceedings of the Fourth Conference on Email and Anti-Spam (CEAS)*. Citeseer, 2007. 121

[141] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *NDSS*, Feb. 2013. 54

[142] J. Lin, N. Sadeh, S. Amini, J. Lindqvist, J. I. Hong, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510. ACM, 2012. 152

[143] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014. 130

[144] M. Lindorfer, S. Volanis, A. Sisto, M. Neugschwandtner, E. Athanasopoulos, F. Maggi, C. Platzer, S. Zanero, and S. Ioannidis. Andradar: fast discovery of android applications in alternative markets. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 51–71. Springer, 2014. 55, 143

[145] H. Lockheimer. Android and Security, Feb 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`. 95, 110

[146] Lookout. Mobile threat report 2011. `https://www.lookout.com/img/images/lookout-mobile-threat-report-2011.pdf`, Aug. 2011. 43

[147] Lookout Mobile Security. State of mobile security 2012. Technical report, sept 2012. 72

[148] K. Lucic. Over 27.44% users root their phone(s) in order to remove built-in apps, are you one of them? `http://www.androidheadlines.com/2014/11/50-users-root-phones-order-remove-built-apps-one.html`, Nov. 2004. 132

[149] A. Ludwig. An update to nexus devices. `http://officialandroid.blogspot.com/2015/08/an-update-to-nexus-devices.html`, Aug. 2015. 23

[150] K. Mahaffey. `http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-a`
Mar. 2011. 19

[151] M. Mangel. *The theoretical biologist's toolbox: quantitative methods for ecology and evolutionary biology*. Cambridge University Press, 2006. 29

[152] A. Marx. Av-test malware repository (collection) statistics. Private data sharing, May 2015. 43, 44

[153] D. Maslennikov. Mobile malware evolution: An overview, part 4. `https://securelist.com/large-slider/36350/mobile-malware-evolution-an-overview-part-4/`, May 2011. 44

[154] D. Maslennikov and A. Gostev. Mobile malware evolution: An overview, part 3. `http://securelist.com/analysis/quarterly-malware-reports/36265/mobile-malware-evolution-an-overview-part-3/`, 2009. 44

[155] D. Maslennikov, A. Gostev, and S. Golovanov. Kaspersky security bulletin 2008: Malware evolution january âĂŞ june 2008. `https://securelist.com/analysis/kaspersky-security-bulletin/36226/kaspersky-security-bulletin-2008-malware-evolution-january-june-2008/`, 2008. 44

[156] L. Menchaca. AT&T sim-locked units and the Froyo update - direct2dell. `http://en.community.dell.com/dell-blogs/Direct2Dell/b/direct2dell/archive/2010/12/09/at-amp-t-sim-locked-units-and-the-froyo-update.aspx`, Dec. 2010. 14

[157] B. Molen. "android 4.0 ice cream sandwich now official, includes revamped design, enhancements galore". Oct 2011. 56

[158] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, (4):33–39, 2003. 28

[159] T. Moore and B. Edelman. Measuring the perpetrators and funders of typosquatting. *Financial Cryptography and Data Security*, pages 175–191, 2010. 127, 135

[160] T. Moore, N. Leontiadis, and N. Christin. Fashion crimes: trending-term exploitation on the web. In *Proc. ACM CCS*, pages 455–466, Chicago, IL, 2011. 119

[161] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, 2007. 95

[162] D. Mutz, G. Vigna, and R. Kemmerer. An experience developing an ids stimulator for the black-box testing of network intrusion detection systems. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 374–383. IEEE, 2003. 95

[163] T. Myerson. Announcing windows update for business. May 2015. 24

[164] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *CCS*, pages 328–332. ACM, 2010. 67

[165] J. Networks. Juniper networks third annual mobile threats report. `http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2012-mobile-threats-report.pdf`, June 2013. 44

[166] J. v. Neumann and A. W. Burks. Theory of self-reproducing automata. 1966. 30

[167] P. Nickinson. `http://m.androidcentral.com/rovio-explains-why-angry-birds-update-needs-sms-`
Feb. 2011. 61

[168] J. Oberheide. Remote kill and install on google android. `http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/`. 9, 20, 23

[169] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012. 95, 110

[170] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in android. pages 221–230, 2010. 67

[171] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. pages 340–349. IEEE, 2009. 59, 67

[172] T. Ooura. Improvement of the pi calculation algorithm and implementation of fast multiple precision computation. *Transactions-Japan Society for Industrial and Applied Mathematics*, 9(4):165–172, 1999. 100

[173] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proc. WOOT*, volume 41, page 86. USENIX, 2009. 94

[174] T. Paraskevakos. Apparatus for generating and transmitting digital information, May 1974. US Patent 3,812,296. 31

[175] B. S. Paul McDonald, Matt Mohebbi. Comparing google consumer surveys to existing probability and non-probability based internet surveys. `http://www.google.com/insights/consumersurveys/static/consumer_surveys_whitepaper.pdf`, Mar. 2012. 128, 158

[176] P. Pearce, V. Dave, C. Grier, K. Levchenko, S. Guha, D. McCoy, V. Paxson, S. Savage, and G. M. Voelker. Characterizing large-scale click fraud in zeroaccess. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 141–152. ACM, 2014. 28

[177] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. ACM, 2012. 67

[178] N. J. Percoco and S. Schulte. Adventures in bouncerland. *Black Hat USA*, 2012. 95, 110

[179] J. Phillips. Path social media app uploads ios address books to its servers. Feb. 2012. 43

[180] H. Pilz and S. Schindler. Are free android virus scanners any good? AVTEST Report, Nov. 2011. 39

[181] B. Prince. Google android trojan, fbi raid linked to operation payback lead security news. `http://www.eweek.com/c/a/Security/Google-Android-Trojan-FBI-Raid-Linked-to-Operation-Payback-Lead-News-406931/`, Jan. 2011. 19

[182] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, DTIC Document, 1998. 95

[183] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *Information Security*. Springer, 2007. 95, 96, 99, 104

[184] J. Raphael. Android 2.2 upgrade list: Is your phone getting Froyo? - Computerworld Blogs. `http://blogs.computerworld.com/16310/android_22_upgrade_list`, June 2010. 14

[185] G. Research and A. Team. Kaspersky security bulletin 2013. malware evolution. 2013. 29, 43

[186] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA, pp229–238*, 1999. 89

[187] M. Rogers. The bearer of badnews. `https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/`, Apr. 2013. 9

[188] C. Rossow, C. Dietrich, H. Bos, L. Cavallaro, M. van Steen, F. Freiling, and N. Pohlmann. Sandnet: Network traffic analysis of malicious software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 78–88. ACM, 2011. 72

[189] F. Ruiz. 'Android/NotCompatible' Looks Like Piece of PC Botnet. `http://blogs.mcafee.com/mcafee-labs/androidnotcompatible-looks-like-piece-of-pc-botnet`, May 2012. 49, 85, 86

[190] J. Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. *Invisible Things*, 2004. 94, 95

[191] N. Saint. 50% of android apps with internet access that ask for your location send it to advertisers. `http://www.businessinsider.com/50-of-android-apps-that-ask-for-your-location-send-it-to-advertisers-2010-10`, Oct. 2010. 60

[192] D. Salmi. Is google protecting me after all? `https://blog.avast.com/2012/12/13/is-google-protecting-me-after-all/`, Dec. 2012. 43

[193] D. Salmi. avast! mobile security trusted by millions to fight android malware. `https://blog.avast.com/2013/09/page/2/`, 2013. 44

[194] A. Schmidt, R. Bye, H. Schmidt, J. Clausen, O. Kiraz, K. Yuksel, S. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *ICC*, pages 1–5. IEEE, 2009. 67, 125

[195] A. Schmidt, J. Clausen, A. Camtepe, and S. Albayrak. Detecting symbian os malware through static function call analysis. In *Proc. MALWARE*, pages 15–22. IEEE, 2009. 125

[196] A. Schmidt, H. Schmidt, J. Clausen, K. YÃijksel, O. Kiraz, A. Camtepe, and S. Albayrak. Enhancing security of linux-based android devices. In *15th International Linux Kongress, Lehmann*, 2008. 67

[197] M. Schönefeld. Reconstructing dalvik applications. *CANSECWEST 2009*, Mar. 2009. 119

[198] M. Schroeder and J. Saltzer. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. 58, 60, 151

[199] M. Schwarts. Google removes malware apps from android market, June 2011. 117, 118

[200] L. C. Scott Keeter. A comparison of results from surveys by the pew research center and google consumer surveys. `http://www.people-press.org/files/legacy-pdf/11-7-12%20Google%20Methodology%20paper.pdf`, nov 2012. 128, 158

[201] N. Seriot. iPhone privacy. Arlington, VA, 2010. Blackhat DC. 68

[202] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-Powered mobile devices using SELinux. *IEEE Security and Privacy*, 2009. 24, 67

[203] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *Security & Privacy, IEEE*, 8(2):35–44, 2010. 8, 9, 17, 20, 59, 67, 76

[204] Y. Shao, X. Luo, and C. Qian. Rootguard: Protecting rooted android phones. *Computer*, 47(6):32–40, 2014. 132

[205] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. Towards formal analysis of the Permission-Based security model for android. pages 87–92. IEEE, 2009. 59, 67

[206] G. Simpson. The major features of evolution. 1953. 45

[207] A. Smith. Chapter one: A portrait of smartphone ownership. `http://www.pewinternet.org/2015/04/01/chapter-one-a-portrait-of-smartphone-ownership/`, Apr. 2015. 8

[208] S. Stanley. Effects of competition on rates of evolution with special references to bivalve molluscs and mammals. *Sysem, Zool.*, (22):486–508, 1973. 45

[209] J. Stewart. Truman-the reusable unknown malware analysis net. *http://www. secureworks. com/research/tools/truma. html*, 2006. 81

[210] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The underground economy of spam: A botmasterâĂŹs perspective of coordinating large-scale spam campaigns. In *USENIX workshop on large-scale exploits and emergent threats (LEET)*, volume 29, 2011. 48

[211] T. Strazzere. Dex education 201 anti-emulation, Sept 2013. `http://hitcon.org/2013/download/Tim%20Strazzere%20-%20DexEducation.pdf`. 96

[212] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX Security Symposium*, pages 399–416, 2009. 151

[213] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerabilty discovery*. Addison-Wesley Professional, 2007. 72

[214] D. S. T. R. Team. Dell network secuirty threat report 2013. `http://www.sonicwall.com/documents/dell-network-security-threat-report-2013-whitepaper-30197.pdf`, Feb. 2014. 44

[215] T. O. Tengs, M. E. Adams, J. S. Pliskin, D. G. Safran, J. E. Siegel, M. C. Weinstein, and J. D. Graham. Five-hundred life-saving interventions and their cost-effectiveness. *Risk analysis*, 15(3):369–390, 1995. 151

[216] D. R. Thomas, A. R. Beresford, and A. Rice. Security metrics for the android ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 87–98. ACM, 2015. 14, 15, 16

[217] K. Tofel. Htc says monthly android security updates are unrealistic. Oct. 2015. 23

[218] Trendmicro. Android under siege: Popularity comes at a price. `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/`

rpt-3q-2012-security-roundup-android-under-siege-popularity-comes-at-a-price.pdf, Oct. 2012. 43

[219] Trendmicro. Mobile threats go full throttle: Device flaws lead to risky trail. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-2q-2013-trendlabs-security-roundup.pdf, Aug. 2013. 43

[220] R. Unuchek. The android trojan svpeng now capable of mobile phishing. https://securelist.com/blog/research/57301/the-android-trojan-svpeng-now-capable-of-mobile-phishing/, Nov. 2013. 52

[221] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999. 78, 92

[222] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. *Sable Technical Report 1998-4, Sable Research Group, McGill University*, 1998. 78

[223] T. Vennon. Android malware. A study of known and potential malware threats. Technical report, Technical Report White paper, SMobile Global Threat Center, Feb. 2010. 10

[224] T. Vennon and D. Stroop. Threat analysis of the android market. Technical report, Tech. rep., SMobile Systems, 2010, June 2010. 60, 67

[225] T. Vidas and N. Christin. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 197–208. ACM, 2013. 29, 35, 39, 40, 41, 47, 70, 72, 117, 143

[226] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. *Proc. AsiaCCS*, 2014. 70, 81, 101, 102, 103, 106, 112

[227] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *Proceedings of the 5th workshop on Web 2.0 Security and Privacy (W2SP)*, volume 2, 2011. 9, 58, 59, 62, 63, 64, 77

[228] T. Vidas, E. Owusu, S. Wang, C. Zeng, L. F. Cranor, and N. Christin. Qrishing: The susceptibility of smartphone users to qr code phishing attacks. In *Financial Cryptography and Data Security*, pages 52–69. Springer, 2013. 35, 57

[229] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 39–50. ACM, 2014. 70, 75, 80

[230] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Workshop on Offensive technologies*. USENIX Association, 2011. 7, 13, 14, 23, 27, 33, 143

[231] T. Vidas, C. Zhang, and N. Christin. Toward a general collection methodology for android devices. *digital investigation*, 8:S14–S24, 2011. 17, 20, 21, 22, 104, 107

[232] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233. ACM, 2014. 76, 99

[233] VirusTotal. VirusTotal - Free Online Virus, Malware and URL Scanner. `https://www.virustotal.com/`. 39, 76

[234] A. Waqas. Root any android device and samsung captivate with super one-click app. `http://www.addictivetips.com/mobile/root-any-android-device-and-samsung-captivate-with-super-one-click-app/`, Oct. 2010. 21

[235] M. Ward. Cryptolocker victims to get files back for free. `http://www.bbc.com/news/technology-28661463`, Aug. 2014. 28

[236] C. Welch. Google: Android app downloads have crossed 50 billion, over 1m apps in play. `http://www.theverge.com/2013/7/24/4553010/google-50-billion-android-app-downloads-1m-apps-available`, 2013. 9

[237] D. Wendlandt, D. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008. 145, 152

[238] L. Whitney. Android market share stays steady in us but sinks deeper in europe. `http://www.cnet.com/news/android-market-share-stays-steady-in-us-but-sinks-deeper-in-europe/`, 2015. 70

[239] Wicherski, G. code.mwcollect.org. http://www.mwcollect.org, Feb 2011. 76

[240] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, 5(2):32–39, 2007. 72, 95

[241] T. Wimberly. Sprint releases android 2.1 for samsung moment. `http://androidandme.com/2010/05/news/sprint-releases-android-2-1-for-samsung-moment/`, May 2010. 14

[242] T. Wimberly. Top 10 android phones, best selling get software updates first. `http://androidandme.com/2010/11/news/top-10-android-phones-best-selling-get-software-updates-first/`, Nov. 2010. 13

[243] J. Wise. Unrevoked3 recovery reflash tool. `http://unrevoked.com/rootwiki/doku.php/public/unrevoked3`, Jan. 2011. 20

[244] B. Woods. Researchers expose android webkit browser exploit. `http://www.zdnet.co.uk/news/security-threats/2010/11/08/researchers-expose-android-webkit-browser-exploit-40090787/`, Nov. 2010. 20

[245] Y. Zhang, S. Egelman, L. Cranor, and J. Hong. Phinding phish: Evaluating anti-phishing tools. In *In Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007. 152

[246] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM SPSM*, 2012. 93

[247] X. Zhou, W. Zhou Y. Jiang and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. 3rd ACM Conference on Data and Application Security and Privacy*, 2012. 55

[248] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012. 40, 48, 55, 70, 72, 81, 89, 92, 123

[249] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012. 29, 35, 47, 55, 76, 99, 143