Securing Decentralized Software Installation and Updates

By

David Barrera

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

Carleton University
Ottawa, Ontario, Canada

©2014, David Barrera

Abstract

Software installation and updates have become simpler for end users in recent years. The commoditization of the internet has enabled users to obtain more software than ever before from more developers than ever before. Additionally, software installation now requires less input from users, allowing installation with merely a single click or tap. At the same time, different software installation models with varying levels of security, usability and freedom have emerged. In centralized environments, available software is limited by an authority, whereas decentralized environments allow users and developers to interact freely. Decentralized software installation ecosystems pose the most significant security challenges due to the lack of centralized control.

In this thesis we identify, through the systematic evaluation of prominent systems, limitations in the way operating systems provide security guarantees (including verification of integrity and authentication, and establishment of trust) in decentralized software installation environments. We address these limitations by designing tools and protocols that help secure software installation and updates at each of the three installation stages (software discovery, initial install and updates, and enforcing security policies). Specifically, we propose a cryptographically verifiable protocol for developers to delegate digital signature privileges to other certificates (possibly owned by other developers) without requiring a centrally trusted public key infrastructure. Our proposal allows trust to be delegated during software updates without user involvement. We also propose a flexible policy for developers to authenticate and share privileges amongst applications being executed simultaneously on a device. We evaluate these proposals and show that they are direct improvements over currently deployed real-world systems.

We discuss the design and implementation of an install-time architecture that allows users to query crowdsourced expert information sources to gain trust in software they are about to install. We motivate the requirements for such a system, designed to mirror the security semantics provided by centralized environments.

The proposed protocols and tools have been implemented as proofs-of-concept using Google's Android mobile operating system. We leverage a large application dataset to inform our design decisions and demonstrate backward compatibility with existing applications. While the implementations are specific to Android, we discuss how our general proposals extend to other decentralized environments.

Acknowledgements

I'd like to thank my advisor, Dr. Paul C. van Oorschot for his guidance, feedback, advice and support during my time (7 years!) as a graduate student. Throughout my degrees, Paul always kept me on track, helping me make the best use of my time. Paul's knowledge and style have, without a doubt, helped me become a better researcher, writer, peer, and mentor.

I owe thanks to my wife Elizabeth for her love and patience during the many (and often stressful) milestones of my doctorate. Elizabeth was a constant reminder that there was life outside the university walls.

I'd like to thank my mom for always checking in on me despite living 3600 Km away. It is always comforting to know that my mom is thinking about me, no matter how far I am. I also owe thanks to my father (the other Dr. Barrera) for always asking about the details of my research and showing great interest in my progress as an academic. And of course to my sisters who always provided their unconditional support.

I'm grateful to Daniel McCarney, Jeremy Clark, and William Enck who helped with the papers that eventually made up this dissertation. I could not have produced this document without their hard work, insight, and discussion. A special mention to Glenn Wurster and Mohammad Mannan for inspiring me when I started graduate school. Finally, I would like to thank all my contemporary Carleton Computer Security Lab members for countless hours of discussion throughout all these years.

Contents

Abstract **iii**

Acknowledgements **v**

Table of Contents **vi**

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**
- 1.1 Motivation and Thesis Goals 3
- 1.2 Main Contributions 5
- 1.3 Related Publications 6
- 1.4 Definitions and Scope 8
- 1.5 Organization 9

2 Background and Related Work **11**
- 2.1 Introduction 11
- 2.2 Deconstructing Software Installation 11
- 2.3 Software Installation Models 13
  - 2.3.1 Centralized 14
  - 2.3.2 Decentralized 15
  - 2.3.3 Hybrid 16
- 2.4 Pre-installation Security Mechanisms 17
  - 2.4.1 Application Vetting 18
  - 2.4.2 Static Code Analysis and Code Review 20
  - 2.4.3 Metadata Analysis 21
- 2.5 Post-installation Security Mechanisms 22
  - 2.5.1 Privacy Leaks 23

2.5.2 Privilege Escalation 23
2.5.3 Malware Scanning 24
2.5.4 Kill Switches 25
2.6 Secure Software Distribution and Updates 26
2.6.1 Secure Software Distribution 27
2.6.2 Secure Software Updates 28
2.7 Summary 29

3 Android Software Installation Security 31
3.1 Introduction 31
3.2 Android Security Model 31
3.3 Application Packages 33
3.4 Permission Model 35
3.5 Digital Signatures 37
3.5.1 Certificate Sets 38
3.6 Secure application interaction 39
3.6.1 UID sharing 39
3.6.2 Signature permissions 40
3.7 Installation, Updates and Removal 41
3.7.1 Installation with the Google Centralized Store 41
3.7.2 Installation without the Google Centralized Store (Sideloading) 42
3.7.3 Updates 43
3.7.4 System workflow 44
3.8 Summary 46

4 Securing Software Discovery 47
4.1 Introduction 47
4.2 Background 48
4.2.1 Application Markets 49
4.2.2 Package Signing and Application Namespace 50
4.2.3 Malware Mitigation 51
4.3 Threat Model 52
4.4 Meteor: Enhancing the Security of Multi-market Platforms 53
4.4.1 Universal Application Identifiers 55
4.4.2 Developer Registries 56
4.4.3 Application Databases 58
4.4.4 Kill Switch Authorities 61
4.5 Implementation of the Meteorite Client Application 62
4.5.1 Information Source Management 62
4.5.2 Information Source Query 64
4.5.3 Kill Switch Listener 64
4.5.4 Design Choices 66
4.6 Discussion of the Meteor Architecture 66

4.6.1 Advantages . . . 66
4.6.2 Limitations . . . 67
4.6.3 Security and Privacy Considerations . . . 68
4.7 Security Analysis . . . 69
4.8 Related Work . . . 70
4.9 Summary . . . 71

5 The Android Observatory **73**
5.1 Introduction . . . 73
5.2 Overview . . . 75
5.3 Application Sources . . . 77
5.4 Populating the Observatory Dataset . . . 81
5.5 Future Work . . . 82

6 Securing Software Downloads and Updates **83**
6.1 Introduction . . . 83
6.2 Background and Related Work . . . 85
6.2.1 Download Integrity . . . 85
6.2.2 Digital Signatures for Authenticating and Verifying Downloads 86
6.2.3 Trust on First Use in Decentralized Environments . . . 87
6.2.4 Selective Updates . . . 88
6.2.5 Other Related Work . . . 88
6.3 Motivation for Key Agility on Android . . . 89
6.3.1 Absence of Secure Defaults . . . 89
6.3.2 Ownership Transfer . . . 90
6.3.3 Logical Requirements . . . 91
6.3.4 Case Studies . . . 92
6.4 Design and Implementation of Baton . . . 93
6.4.1 Threat Model and Goals . . . 96
6.4.2 Implementation . . . 96
6.5 Evaluation . . . 101
6.5.1 Compatibility . . . 101
6.5.2 Implementation Evaluation . . . 103
6.5.3 Security Analysis . . . 104
6.6 Discussion . . . 105
6.6.1 Limitations of Existing Proposals . . . 105
6.6.2 Certificate Expiration . . . 106
6.6.3 Private Key Compromise . . . 107
6.6.4 Transferring Authority . . . 107
6.6.5 Applicability Beyond Android . . . 109
6.6.6 Limitations of Baton . . . 109
6.7 Alternative Approaches . . . 110
6.7.1 Centralized Private Key Infrastructure . . . 110

6.7.2 Certificate Trees 111
6.7.3 Distributed and Threshold Signatures 111
6.8 Summary 113

7 Applying and Enforcing Application Security Policies 115
7.1 Introduction 115
7.2 Security Policies in Decentralized Environments 116
7.2.1 Developer Policies 116
7.2.2 Guardian Policies 117
7.2.3 User-configured Policies 118
7.2.4 OS Vendors 118
7.3 Android Policies for UID Sharing 120
7.3.1 Permission Inheritance through UID Sharing 123
7.4 Improving Android's UID Sharing Policies 124
7.4.1 Alternative Mechanisms for UID Sharing 126
7.4.2 Implementing Fluid UID Sharing 127
7.4.3 Discussion 128
7.5 Related Work 131
7.6 Summary 132

8 Discussion and Conclusions 133
8.1 Revisiting Thesis Goals 133
8.2 Open Problems in Decentralized Software Installation 134
8.3 Summary 135

References 137

Appendices 148

A Cognitive Walkthrough: Authenticator 149
A.1 Walkthrough Setup 149
A.2 Evaluation 150
A.3 Interpretation of Results 152

B List of Android Permissions 153

List of Tables

4.1 Types of entries in a Meteor application database 57
4.2 Example application database 59

6.1 Summary of signing key use on the Android Observatory 90
6.2 Comparison of certificate agility proposals 106

7.1 10 most-used sharedUID strings on the Android Observatory 121
7.2 Comparison of UID sharing mechanisms 126

List of Figures

2.1 Example FreeBSD software installation 17

3.1 Android software stack 32
3.2 Example code demonstrating world readable file creation 33
3.3 Contents of an Android application 34
3.4 Permission requests in Android applications 36
3.5 Android OS signature verification method (pre-2.3) 38
3.6 Android OS signature verification (2.3 and later) 38
3.7 Android app installation flow screen captures 42
3.8 Android app updates screen captures 43
3.9 Android's software installation workflow 44

4.1 Overview of the Meteor architecture 53
4.2 Overview: Meteor 62
4.3 Adding an information source to the Meteorite app 63
4.4 Meteor server manifest example 63
4.5 Screenshots of the Meteorite app 65

5.1 Android Observatory statistics 74
5.2 Android Observatory search 74
5.3 Android Observatory upload 76
5.4 Android Observatory application sources 78
5.5 Block diagram of application import process. 81

6.1 Verifying integrity of a downloaded file 85
6.2 CDF for certificate reuse on the Android Observatory 90
6.3 Comparison of Baton and stock Android (various update conditions) 95
6.4 Example Baton certificate chain entry 98

7.1 CyanogenMod's AppOps interface 119
7.2 Permission inheritance through UID sharing 122

7.3 Example Fluid-enabled Android manifest (root application) 129
7.4 Example Fluid-enabled Android manifest (leaf application) 129

Chapter 1

Introduction

Modern computing platforms such as personal computers, smartphones and tablets offer incredible amounts of processing power, data storage and access to detailed environmental readings (*e.g.,* via Global Positioning System (GPS) sensors, cameras, and gyroscopes). Operating systems running on these modern platforms, in their initial (default) configurations, provide software applications which make basic use of these advanced features. However, users are typically directed to third-party[1] software if they wish to fully utilize all the features on their devices, or apply them to novel uses.

The way users obtain and update third-party software for their devices has changed significantly over the past few decades. The internet has provided a viable alternative to (and in many cases entirely replaced) physical installation media (*e.g.,* floppy disks and CD-ROMs), allowing network-based software installations from diverse sources. The following factors have also motivated the transition to network-based software installations:

Smaller physical devices. As devices become increasingly mobile, manufactures of smaller devices are removing expandable media slots such as CD-ROM bays, memory card slots, and Universal Serial Bus (USB) ports. The absence of these slots encourages (and sometimes limits) software to being installed over the network.

Reduction of costs. The economic cost of producing physical installation media is

[1] Third-party software is installed and updated alongside the base operating system software load, but may be developed by parties other than the OS vendor, *e.g.,* an independent developer or group.

high compared to the cost of hosting an installation package on a web server. Hosting costs can additionally be very low or even free if they are sponsored by a platform vendor or subsidized by advertisement revenue. Time costs are also a factor when considering delivery and shipping times as compared to making a digital download immediately available, possibly to a worldwide userbase.

Need for updates. If software flaws are discovered after initial release, or missing features need to be added, software updates can be distributed online. Updates to software distributed on physical media rarely reach the full user base[2], while software initially distributed through online sources may more easily receive frequent updates.

Prior to the internet, acquiring software required a physical distribution channel. Purchasing software on diskettes or CDs at a retail store was common, as were offline peer to peer software transfer schemes such as *sneakernet*[3] for sharing software amongst friends and special interest groups. The periodic installation of software in corporate environments through disk imaging or similar techniques was also common, giving employees updated copies of software every few months. As software binaries and installation packages grew in size (*e.g.,* from small file size text-based games to complex 3D games with intense graphics and textures), optical media such as CDs and DVDs became the norm for software distribution. Later, the commoditization of the internet allowed applications which were small in size to be downloaded from remote servers, but larger applications such as office suites and games continued to rely on physical media distribution. Still, as of writing, physical media persists as the preferred method for distributing large applications such as video games for users who do not have access to large amounts of bandwidth.

Ease of software installation has also increased in the past decades. Today's software, particularly software which targets modern operating systems, can be installed and updated with a single click or touch of a screen, whereas installation previously required obtaining physical media, loading it onto the system and following a set of prompts. While the increased usability properties of software installation are welcome, they present a problematic scenario when combined with increased exposure to a large software base: users today have *more access to software* than ever before,

[2]Hybrid update models exist, where the initial installation is provided via physical media, and updates such as service packs or expansion packs are distributed online.

[3]The transfer of electronic information by physically moving the storage media from one location to another.

written by *more developers* than ever before. Single-click installations from centralized application repositories (*i.e.,* app stores) make discovering and installing software trivially simple for users, while simultaneously creating an environment that is ripe for exploitation; as users become accustomed to installing more software without any perceived negative impact, concerns about security become lower priority or security is simply assumed to be present.

On desktop operating systems, users have gravitated towards a web-centric model, where a large portion of computing time is spent visiting different websites and services inside a web browser. When a web service is not available, downloadable applications are used. Smartphones and tablets on the other hand have moved towards discrete, task-specific installable applications (commonly referred to as *apps*), likely for two reasons: (1) web browsers on mobile are not as feature-complete as their desktop counterparts; and (2) the smaller user interface encourages carrying out targeted tasks with minimal user input. The app culture has led to many users installing dozens and in some cases hundreds of apps on their devices.

App installation has evolved into several models, each providing security, usability and freedom properties. *Centralized* models where a central party (*e.g.,* software or hardware vendor) has full decision control over what applications are made available to users are believed to provide high security and usability at a high cost of freedom. *Decentralized* models, where users can install software from any developer without restriction are often considered to offer weaker security guarantees but give users more alternatives, encouraging unrestricted software development and consumption.

1.1 Motivation and Thesis Goals

We believe the new software installation paradigm of users frequently installing many applications from a wide range of developers motivates the need for security mechanisms and supporting infrastructure to enhance installation and updates. As users trust their devices and personal data to more developers with each new installation, it is unreasonable to expect that end-users will be successful in distinguishing benign from malicious developers and applications, especially at a large scale. We believe there is also motivation to create security architectures that isolate (both from the

OS and from each other) untrusted applications and limit applications' capabilities once installed. The design and implementation of these security mechanisms is the overall motivation of this thesis. Specifically, the research and proposals made in this thesis are driven by the following goals:

G1. Improve Decentralized Software Installation Security Mechanisms.

We identify limitations in currently deployed decentralized OS software security mechanisms related to software installation. When limitations are found, we propose improvements to make such mechanisms more robust and less prone to error when used by non-technical users. Our goal is to provide improvements which can be incrementally deployed to devices while maintaining backwards compatibility[4]. We inform our proposals by empirically and systematically analyzing designs of currently deployed systems. We specifically focus on issues related to establishing trust relationships between applications written by multiple developers, as well as transparent trust delegation during software updates.

Each of our proposals is designed with the goal of minimizing involvement from the end-user (*e.g.,* not asking the user for input unless absolutely necessary). While we aim to propose generic improvements that are applicable to many (hopefully all) decentralized platforms, we implement and evaluate concrete proposals on the open source version of Google's Android operating system [70]. Where possible, we discuss the suitability of extending our improvements to other platforms as well.

G2. Propose New Architectures to Enhance the Security of Decentralized Software Installation.

Where incremental changes improving on current security mechanisms are not feasible, we propose entirely new mechanisms or architectures. These proposals may require replacing major components in existing OSs, or require server-side supporting infrastructure. Our objective here is that future OS designers (particularly those in decentralized environments) will consider our proposals for inclusion (either directly or variants) into their overall OS software installation security architecture. We focus

[4] Throughout the thesis, we use the term *backwards compatibility* to describe not needing to change all currently existing applications in order to maintain interoperability with our changes.

on the problem of establishing an initial trust relationship between end users and developers in the absence of a central authority.

1.2 Main Contributions

In this thesis, we discuss mechanisms for securing the software installation process in decentralized environments. To that end, our first contribution is an in-depth analysis of the three main components of the software installation process:

1. Software discovery and initial trust establishment. We explore how users can find trusted software applications in the presence of multiple, possibly untrusted, application repositories. We identify requirements for achieving security semantics of single-market environments when there are multiple markets available.

2. Integrity verification and trust continuity. We perform an in-depth analysis of the Android software update process, including the secure retrieval of a copy of the software from an installation source, and possibly overwriting a previously installed copy as an update. Here we discuss challenges of authentication in decentralized environments and propose a mechanism to prevent applications from being overwritten by unauthorized parties. We also present a mechanism for cryptographically verifiable trust delegation.

3. Security policy enforcement. We analyze different approaches for distributing, applying, and enforcing application security policies in the absence of a central authority. For Android (versions 2.0 to 4.4) specifically, we identify limitations in the granularity of privilege sharing between applications. We propose an alternate mechanism to specify applications with which a given application may share privileges.

In addition, this thesis also discusses the design, implementation and evaluation of four software tools developed as proofs-of-concept.

Meteor. Meteor is a software installation framework designed to assist users in identifying trusted applications on initial install. Meteor is comprised of a client application that interfaces with multiple server back-ends. The client queries the server for additional relevant information about an application to be installed. The server can notify the software client (called Meteorite) that an application is known to be

malicious, or point the user to alternative applications written by the same or other developers. This may help users find more suitable software alternatives. Meteorite clients may optionally register to receive kill switches (app removal requests) from the server in the event that a server operator finds an application to be malicious. The feasibility of Meteor is demonstrated by building and testing the client software on Android and the server on Linux.

The Android Observatory. The Android Observatory is a publicly available website and service that allows users to view detailed information about Android applications. The Observatory processes Android applications extracting metadata and finds similarities to other applications such as other apps signed with the same key, same binary resources, permissions, *etc.*. As of writing, the Android Observatory has indexed over 53,000 Android applications. The Observatory receives dozens of volunteer app submissions per week. We have also shared the database of parsed application metadata with several other international research groups.

Baton. Baton is a protocol that allows developers to delegate signing authority to a new signing key and corresponding digital certificate. We implement Baton on Android, where it was not previously technically possible to replace or make changes to the signing key used to sign the initial release of an application. Baton allows signing key updates without requiring a central authority of any kind. Baton is also designed to be backwards compatible with existing applications and Android versions.

Fluid. Fluid is an alternate method for two or more Android applications to share privileges, process, or memory space. Prior to this proposal, the only way to achieve this level of integration was for developers to sign all applications in the (desired sharing) group with the same signing key. Our proposal allows developers to retain their own signing keys, while explicitly authorizing applications in a policy. The shared privileges can be revoked (through an application update) at a later time if necessary.

1.3 Related Publications

Most of Chapter 2 was published as a periodical article on software installation. For this thesis, the content thereof was expanded and updated, removing focus on

smartphones and generalizing the software installation models to modern operating systems.

- D. Barrera and P.C. van Oorschot. Secure Software Installation on Smartphones. *IEEE Security & Privacy*, 9(3):42-48, May 2011.

The software installation infrastructure (Meteor) presented in Chapter 4 was designed in conjunction with Dr. William Enck, who was instrumental in discussing the overall concept and security design. Most of the writing in the subsequent publication of the work is my own:

- D. Barrera, W. Enck, and P.C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *IEEE Mobile Security Technologies Workshop (MoST)*, 10 pages, 2012.

Chapters 3 and 7 include text from a peer-reviewed publication written in collaboration with Dr. Jeremy Clark and Daniel McCarney. Clark and McCarney assisted with deconstructing Android's software installation framework and implementing the proposed changes, respectively.

- D. Barrera, J. Clark, D. McCarney, and P.C. van Oorschot. Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 81-92, 2012.

Chapter 6 contains certain portions of the paper listed immediately above, in addition to research done in collaboration with Daniel McCarney and Dr. Jeremy Clark. McCarney was responsible for the implementation and testing of the Baton software, while Clark was instrumental in discussing the design and specifications of the system. The paper was accepted for publication in May 2014:

- D. Barrera, D. McCarney, J. Clark, and P.C. van Oorschot. Baton: Certificate Agility for Android's Decentralized Signing Infrastructure. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. 2014.

Finally, a small portion of Chapter 3 relating to Android's permission model was updated and included from an early publication focusing on analysis of permission-based security models. The paper was written in collaboration with Dr. H.G. Kayacik, but the text used in this thesis is my own.

- D. Barrera, H.G. Kayacik, P.C. van Oorschot, Anil Somayaji. A Methodology for Empirical Analysis of Permission-based Security Models and its Application to Android, In *ACM Conference on Computer and Communications Security (CCS)*, pages 73-84, 2010.

All the publications above were supervised by Dr. Paul van Oorschot, who provided invaluable assistance during the writing process and in shaping each of the papers.

1.4 Definitions and Scope

This thesis focuses on the evaluation and proposal of software installation security mechanisms that do not require a trusted central authority, that is, mechanisms that can improve the security of software installations in decentralized ecosystems.

We use the term *decentralized* to refer to an environment where developers can independently create software and deliver it to users without the need for inclusion in a centralized or formal application store, nor need to obtain a developer license from the OS vendor (the requirement of such a license is the norm in Apple's iOS and Microsoft's Windows Phone, for example).

We use the term *software installation* to refer to the overall process that occurs when software is being installed. While security tools and mechanisms can be used prior to installation (and indeed, much of the overall security of installation depends on secure pre-installation) we do not consider pre-installation as our main focus since it more generally relates to software security instead of installation. Post-installation security mechanisms are also critical to the overall security of a system. However, these are similarly beyond our main scope. Chapter 2 discusses, for completeness, software security mechanisms typically performed prior and post installation.

Android as a Decentralized Ecosystem

Throughout this thesis, we frequently refer to Android as the driving example of a widely deployed decentralized ecosystem. Many Android devices by default include Google's software repository (presently known as *Google Play*). For users who install software only from Google Play, and never from third party sources, the ecosystem

more closely resembles a centralized ecosystem where Google is the trusted authority. In this thesis, our main focus is the case where (Android or other) users install software from third-party application markets (many are available as described in Chapter 5), or directly from the developer as well as possibly from Google Play.

1.5 Organization

The remainder of this thesis is organized as follows. Chapter 2 covers background concepts relating to software installation frameworks and software installation models. This chapter also reviews related work to pre- and post-installation software security mechanisms. In Chapter 3, we perform a comprehensive analysis of Android's software installation framework, focusing on security aspects. In Chapter 4, we discuss security problems with current software discovery in decentralized application ecosystems and propose an improved software installation infrastructure for these environments. In Chapter 5, we discuss the design and implementation of the Android Observatory which can be used as an information source for the architecture described in Chapter 4. Chapter 6 presents a proposal for improving signing key delegation in environments where there is no centralized certificate authorities. We discuss the general protocol, as well as an Android-specific implementation and evaluation. Chapter 7 discusses mechanisms for securing inter-application communication through the use of application security policies. Here we discuss an improvement to Android's application interaction policy which enables trust establishment and revocation as needed. In Chapter 8, we after revisit our thesis goals and discuss remaining open problems in securing decentralized software installation and updates.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter provides necessary definitions and related work to contextualize decentralized software installation. We present definitions that we use throughout this thesis, and further define our scope. This chapter also covers related work in software security, relevant but not specific to installation. Related work specific to each of our proposals is reviewed in subsequent chapters.

2.2 Deconstructing Software Installation

The concept of *software installation* is used broadly throughout the literature. It is often used interchangeably with other similar terms such as software updates, installers, and package management. Generally speaking, software installation refers to the act of making a program ready for local execution. However, depending on the target computing platform, type of software, or intended use, preparing a program for execution may result in different sets of procedures.

For example, software installation can be as simple as retrieving a remote, pre-compiled binary and placing it in a directory where the user has execution privileges. A more complicated example might require the system's software installation

framework — the OS programs and libraries designed to assist in performing software installation — to confirm minimum system requirements are met, identify and resolve shared library dependencies, verify the integrity of each downloaded component, copy necessary documentation and examples, and finally set up paths and links so that all users on the system can run the software. The latter example is representative of Linux package managers (*e.g.,* Debian's APT [98] or Archlinux's Pacman [15]), while the former example typically occurs when downloading simple, standalone applications.

Since this thesis focuses on security, we deconstruct the software installation process into the different stages during which security decisions are made. To keep the narrative focused, throughout the thesis we'll assume the applications being installed are self-contained and thus requires no external software dependencies for execution. We'll also assume that a software update is functionally similar to performing a new installation. However, updates must preserve existing user data. We divide the software installation process into three logical and sequential steps. A brief description of each step is given below, noting that steps 1, 2 and 3 are discussed in detail in Chapters 4, 6, and 7, respectively.

(1) Software Discovery. A user's intent to install an application initiates the software discovery and overall software installation process. Users might compare different applications by vendor, price, user ratings, *etc.* Centralized environments will give users few[1] software installation sources, while decentralized environments will have plentiful alternatives. The software discovery may also be initiated by the system notifying the user that an update for an application is available.

(2) Software Download and Update Integrity. Once the user has selected an application for installation, the application must be securely retrieved (*e.g.,* from a remote server or external file system). Here, integrity checks may verify the file transfer completed without data being modified in transit. In the case of an update, the system or installation framework can ensure that the new version was endorsed (usually through digital signatures) by the developer of the currently installed version. Caution must be taken to ensure user data is not deleted during updates.

(3) Applying and Enforcing Application Security Policies. The final step in the installation and update process is to apply a security policy to the newly installed

[1] Generally only a single option is available.

or updated application. The security policy may have been previously created by the OS vendor, provided by the application developer, or manually configured by the user. Combinations of the previous three options are also possible. The security policy will be applied at install-time and enforced at runtime, ideally preventing unwanted applications from gaining privileges they have not been granted.

We note that the steps listed above exclude any static security analysis of software (*e.g.,* malware scanning) that is done, possibly at a large scale, prior to the installation process. Similarly, once the application is installed, performing runtime security analysis of applications during execution is also not part of the installation process. These two steps are important to software security overall. Thus, related work is discussed for completeness in Sections 2.4 (pre-installation security) and 2.5 (post-installation security), but the steps remain out of scope for this thesis. For the purposes of this thesis, the software installation process begins with user intent to install a program, and ends once the local copy of the application is ready for its first execution.

A note on automatic updates

Many modern operating systems and applications support updating software without user intervention. These automatic updates are sometimes known as silent updates or unattended updates. The general workflow for these updates is to first notify the user of an available update (usually through the application or system periodically polling an update server), then apply the update following the process outlined in step (2) above. The update process will usually require the application in question to be restarted and optionally to notify the user that the application was updated. We include automatic updates in our scope, and assume user intent by enabling the auto-update functionality either in the OS or per-app.

2.3 Software Installation Models

To better understand and compare the different types of software installation models, we find it useful to reference a common framework. To that end, this section describes

a classification of software installation models for third party applications.

We see three generic models for software installation, classified by the level of control the OS or hardware vendor has over third-party software installation and application management. The *centralized* model provides the vendor with the most control. In the *decentralized* model, the vendor has practically no control over software once the device is sold. The *hybrid* model is a middle ground that can be adapted to several environments.

2.3.1 Centralized

The centralized model gives the OS vendor full control over third party installation of software on end-user devices. Users can only install software that has been approved and made available through a vendor's app market or clearinghouse. Due to their restrictive practices, centralized environments are sometimes referred to (perhaps more appropriately) as *closed platforms* or *walled gardens*[2].

In the centralized model, applications can be removed from the clearinghouse by the vendor (sometimes without developer warning), as well as remotely uninstalled or disabled on user's devices (see *Kill Switches* in Section 2.5.4). Code signing is an essential part of this model, since it provides a reliable technical mechanism to provide evidence that an app was sanctioned by the vendor and has not been modified. The walled garden model places most of the security decisions and testing in the vendor's control, giving even non-technical users a (perhaps unfounded) worry-free experience.

While this model is subject to controversy due to the vendor's totalitarian control over the user experience on the device, it provides a strong set of tools to control platform security. In the event that a malicious application is detected (even post-vetting; see Section 2.4.1), the app can be uninstalled from any or all devices that installed the application. The vetting process itself, in conjunction with the single point of software entry on to the device, also allows the vendor to monitor trends and tightly control use of features on user devices.

[2] Walled gardens exist beyond personal computing devices such as PCs, tablets and smartphones. E-book readers, video game consoles, and cellular carriers will often limit devices that can access their network, as well as tightly control content that is delivered to user devices.

Jailbreaking and Installing from Untrusted Sources

Under certain conditions, it is possible for users of devices in centralized environments to remove the restrictions established by the central authority. Once the restrictions are removed, it is possible for users to install software from any source (albeit at their own risk). Vendors in centralized platforms typically make removing software installation restrictions from a centrally managed device difficult by locking the *bootloader* such that only digitally signed images can be booted. The process or removing bootloader locks varies depending on the manufacturer.

A popular example of removing software installation restrictions in a centralized environment is *jailbreaking*[3] Apple's iOS. The first jailbreaks for iOS were made public shortly after the first release of iOS [115] and gave iOS users access to the full device filesystem. This enabled users to bypass codesigning restrictions and directly install software onto the device. Once software installation restrictions are removed, devices behave as decentralized (see below).

2.3.2 Decentralized

In this model, the OS and hardware vendor place no restrictions over what software can be loaded on to devices. Thus, the user is responsible for deciding what software installation sources, developers and applications to trust. Users are free to install software from any source (*e.g.,* a website, memory card, application marketplace), taking on the risk that any or all applications could be malicious since there is no application vetting by default. However, software repositories or third party clearinghouses may independently offer application vetting or certification services.

The decentralized model relies more heavily on strong operating system security features such as application isolation to limit the negative impact of malicious applications on other applications, on the user, and on system software. Additionally, the lack of a central trusted party results in users being responsible for detecting malicious applications after installation.

[3]The term jailbreaking is believed to be in reference to escaping the BSD-like "jail" in which many system applications are executed on iOS.

Prominent examples of decentralized ecosystems such as Google's Android OS have proven that software installation security under this model may require asking users questions which they do not have the technical expertise or detailed knowledge to answer. These questions may be asked either at install time, or at resource access time (*e.g.,* "Do you want to allow this application to read phone state?"). These types of questions have been shown to be largely ineffective with non-technical users [54, 53].

2.3.3 Hybrid

In the hybrid model, some security decisions may be offloaded to a knowledgeable or trusted third party. However, depending on the system, users may still be allowed to install their own software, bypassing the trusted third party. The trusted party has been described by Wurster [126] as a *guardian*.

The guardian's role can be assigned to a variety of entities ranging from the OS vendor (in which case the guardian model behaves more similarly to the centralized model), a mobile phone carrier, an acknowledged expert acting on behalf of a less knowledgeable group of users, or an enterprise system administrator who already controls policy on other employee devices. The guardian is typically in charge of making most of the fundamental security decisions (*e.g.,* which apps are allowed to be installed, what services they are allowed to access on the device); end users are minimally involved with making decisions thereafter. The guardian may also perform a less rigorous application vetting process (*e.g.,* banning applications that violate corporate policy or project values[4]). This method provides a flexible middle ground for software installation that can be fine-tuned according to the required level of security or control.

The hybrid model is common in Linux and BSD distributions, where the distribution vendors (or core developers) release a Linux kernel, core system utilities, and a package manager. The distribution maintainers provide support and updates for this set of applications, while support for third party applications must be provided by upstream[5] developers. Users are also free to download applications from devel-

[4]Debian Linux does not include proprietary (*i.e.,* non-free or open source) applications in its package manager [38].

[5]The term *upstream* is used in the open source community to refer to the original application or

```
Installing irssi-0.8.15_6... done
===> SECURITY REPORT:
      This port has installed the following files which may act as
         network servers and may therefore pose a remote security
         risk to the system.
      /usr/local/bin/irssi

      If there are vulnerabilities in these programs there may be a
         security risk to the system. FreeBSD makes no guarantee
         about the security of ports included in the Ports
         Collection. Please type 'make deinstall' to deinstall the
         port if this is a concern.

      For more information, and contact details about the security
         status of this software, see the following webpage:
      http://www.irssi.org/
```

Figure 2.1: Example installation of the `irssi` IRC client on FreeBSD 10.0. Once the installation has completed, a warning is displayed to the user explaining that third-party applications (ports) may be insecure.

oper's sites directly, albeit at their own risk. Figure 2.1 shows an example message displayed to users when installing a third-party application that includes a network server. FreeBSD developers state that they make no guarantees about the security of these third-party applications, and direct the user to the upstream developer for more information.

2.4 Pre-installation Security Mechanisms

The security mechanisms presented in this thesis are designed to operate during the software installation process. While the operating system and software installation framework make a number of the security decisions during the software installation process, security tools and mechanisms may also be used before and after installation. In this section we review related academic work involving pre-installation security mechanisms. In Section 2.5, we perform a similar review of runtime (post-installation) security mechanisms.

library developer when reporting bugs to distribution maintainers, who are often only responsible for compiling and packaging third-party source code.

2.4.1 Application Vetting

Platform vendors that provide controlled application repositories typically perform some level of vetting on applications submitted for inclusion in their repositories. This section describes some of the tests that vendors run on applications during the app vetting process. We note that many application repository maintainers do not publicly detail their testing process, but often make available to developers documentation outlining test criteria and submission guidelines. We base the following list on documents from Nokia's SymbianSigned Test Criteria [112], Apple's App Store review guidelines [13] and Mozilla's add-on review process [92]. We augment the list with anecdotal reports from developers, as well as our own experience when dealing with application submissions to centralized repositories.

Smoke tests. These tests involve a quick overview inspection of the application to ensure that it does not catastrophically fail. Generally this is not a thorough test, but rather an initial sanity check to verify that the application is worth the full testing process (provided there is one). Smoke tests help filter out broken applications submitted by mistake, as well as poorly written applications. This type of test must be simple to perform in an automated fashion, reducing costs albeit sometimes at the expense of accuracy. It is our understanding that all controlled markets perform at least basic smoke testing on submitted applications.

Hidden API checks. Operating system libraries may expose application programming interfaces (APIs) that are reserved for system applications. These APIs are generally not present in developer documentation, as they are intended to be used only by the OS vendor. Developers sometimes use hidden APIs (by either guessing function names in a common namespace or reverse engineering the OS) to obtain direct access to low-level functionality or to speed up their application by avoiding unnecessary layers of abstraction (especially in graphics and media code). Developer use of hidden APIs is regarded as a poor programming practice, since these APIs could change with future OS releases. Static code analysis and debugging can help identify use of hidden APIs, but all instances might not be revealed. Manual testing and fuzzing may be required for this test.

Functionality checks. These tests verify that the application being submitted is capable of undergoing "typical expected use" without interfering with other installed

applications. Details of how such tests are performed are not always made available by vendors, but we expect manual testing is required. Functionality tests involve simulated real word application use to ensure the application opens, closes, does not crash, etc. Other checks may include verifying that an application does not disrupt basic phone functionality (e.g., the ability to receive a phone call or message) or drain the battery. Some vendors might also perform a more detailed suite of tests on the most popular applications in their application repository.

Intellectual property, liability, and TOS checks. These checks involve verifying that the submitted application does not violate Terms of Service (established by the OS vendor or carrier) or infringe on intellectual property. Tests in this category are usually performed to limit the vendor's liability in the event of a legal dispute surrounding the application once it has been approved. Such checks can be partially automated by looking for specific trademarked keywords or files, but likely require some manual inspection if searching for objectionable[6] content or simply rely on independent notification or complaints by third parties.

User interface checks. Some vendors place heavy emphasis on the user interface of the application in an attempt to deliver a more consistent user experience. For these vendors, testing the user interface (i.e., the placement of buttons, colour schemes, navigation within the application, etc.) is important. Failure to comply with established UI guidelines could result in the application being rejected from a vendor's controlled market. Checks in this category are believed to be done manually rather than in an automated way.

Bandwidth checks. Application use of excessive amounts of bandwidth can severely impact a network (particularly mobile networks). Applications that stream Internet radio or download large files may be further tested to see if they operate within a network operator's infrastructure constraints. The size of the application itself may also be analyzed to determine the best (least costly) method of delivery to users (*e.g.,* WiFi or cellular data).

Security checks. The surveyed application submission guidelines briefly mention malware and trojan horses being unacceptable, as expected. However, there is less detail of how malware checks are performed, if at all. Mozilla's add-on repository

[6] Apple uses an even more subjective term *over the line*, and defines it as: "We'll know when we see it" [13].

contains many extensions that are open source, and Mozilla claims to review source code where possible [92]. When source code is not available, it is unclear what is tested in terms of security. In 2012, Google announced [64] that it was performing dynamic analysis of applications submitted to the Google Play Store. Their dynamic analysis framework is discussed in Section 2.5.

From the list above, it appears that most vetting tests are geared toward reducing liability from the vendor and improving user experience. Security plays a small role during the vetting process. It appears that vendors who control application repositories mainly rely on their ability to remove applications or developers from their repository a posteriori rather than attempt to thoroughly investigate or certify the applications upon submission.

2.4.2 Static Code Analysis and Code Review

Several academic proposals exist regarding statically analyzing applications. Static analysis is performed without running the application in question, in most cases by using automated tools looking for specific code patterns. Code review, as the name implies requires access to the program source code which may not always be available. Thus, decompiling[7] and manual reverse engineering is required.

Egele *et al.* [42] statically analyzed 1,400 iOS applications. The research was conducted by decompiling each application (a Mach-O binary from compiled Objective-C code) and automatically creating a control flow graph (CFG). The CFG is then used in a reachability analysis phase to detect whether there are any data flows between sensitive cell phone identifiers and network API calls. The authors find that Apple's vetting process (as of early 2011) failed to detect many instances of privacy violations, putting user data at risk.

Enck *et al.* [44] performed a large scale analysis of Android applications by decompiling 1,100 popular free applications using a custom decompiler. The authors use

[7] A decompiler is a program that given a binary, attempts to produce a high-level language that performs the same function as the original binary once compiled [33]. Successful decompilation of binaries is a difficult problem, particularly in the presence of obfuscated code. However, general approaches exist with varying levels of success dependent on the language and compiler used, number of subroutines introduced, *etc.* (*cf.* [33]).

static analysis on the resulting 21 million lines of recovered code, and note that while pervasive use of advertisement and tracking libraries exists (present on over 50% of analyzed applications), malware was not present in their dataset. The lack of malware may have been observed due sampling bias, as only the most popular applications from the official Google Play Store were included in the dataset.

Work has also been done attempting to statically create a mapping of Android API calls to Android permissions. As of writing, there is no automated way for Android developers to determine the set of permissions needed for an application to run. Thus, developers must read the relevant documentation (which is sometimes incorrect or outdated) and manually assert needed permissions. Felt *et al.* [51] built a tool to scan compiled Android applications in an effort to automatically detect over-declaration of permissions (*i.e.,* permissions requested but never used). Similar work by Au *et al.* [17], produced a permission-to-API calls map. Au *et al.*'s methodology differs from Felt *et al.* in that the former authors generate call graphs on the full Android source code, whereas the latter look at call graphs on individual applications.

2.4.3 Metadata Analysis

Software may be packaged and distributed with additional information about the application (*e.g.,* related to Android, see Section 3.3). This *metadata*, such as name, version number, permissions, author, *etc.*, may help users learn more about the application without installing it. Of course, the authenticity and integrity of application metadata should be protected to prevent modification which might result in tricking users into installing unexpected versions or modified copies of software. This section discusses academic proposals to analyze application metadata with a focus on software security.

Kirin [45] suggest that permission (a complete list of Android permissions is provided in Appendix B) combinations might be dangerous. For example, permissions to write to the SMS database (`WRITE_SMS`) and receive incoming messages (`RECEIVE_SMS`) could give an application the ability read incoming messages and delete them, or modify them before writing.

Barrera *et al.* [21] used the self-organizing maps machine learning technique to visu-

alize Android permission requests (see Section 3.4) in the top 1,100 Android applications on the Google Play store in December 2009. The authors find that there is no correlation between declared application categories (which are developer chosen) and permission requests made by the application. This observation becomes relevant when users are expected to interpret permission requests and decide whether to trust an application. The authors also find that while Android's permission system is very granular, the use of permissions by application developers is not uniform; a small number of permissions (*e.g.,* those restricting access to network connectivity, GPS access, and SMS functionality) are used by a large percentage of applications, while most permissions are rarely or never used.

Metadata such as app price, reviews, number of downloads and ratings, all of which are provided by application markets, can also be analyzed with data-mining techniques. Chia *et al.* [30] performed statistical analysis comparing permission requests with app popularity. For their analysis, the authors collected large datasets of Android applications from multiple markets, Facebook applications, and Google Chrome browser extensions. The authors find that even across multiple platforms and versions, popular applications request more permissions (*i.e.,* are more privileged). Chia *et al.* also find that developers often modify the branding of their application (*e.g.,* app name and description) to mislead users into installing applications which are believed to provide some level of functionality.

2.5 Post-installation Security Mechanisms

This section briefly covers related work in proposing security mechanisms whose operations require the application to be executed. In general, the work below involves the dynamic analysis of applications to identify privacy leaks, unauthorized privilege escalation, and malicious program components.

2.5.1 Privacy Leaks

On the defensive side, TaintDroid is an Android modification designed by Enck *et al.* [43] to detect privacy leaks in applications. While taint tracking has previously been used in desktop operating systems to detect attacks, Enck *et al.* were the first to develop a dynamic taint analysis solution for a mobile platform where resources are limited. TaintDroid works by flagging sources of sensitive data (*e.g.,* cell phone identifiers, contacts, and messages) when relevant API calls are made, and identifying flagged data if and when it reaches a *sink* (usually a network interface). Since TaintDroid requires deep system integration, it must replace the local Android base operating system installation. However, TaintDroid may allow experts to test third-party applications for privacy violations.

On the attack side, Schlegel *et al.* [108] demonstrate a trojan for Android that is capable of identifying sensitive information spoken into the device's microphone (*e.g.,* credit card numbers and PINs). The trojan requires the installation of two separate low-privileged applications that communicate through covert channels on the device once installed.

2.5.2 Privilege Escalation

A heavily researched area in OS security focuses on attacks and defences against privilege escalation. A privilege escalation attack usually involves an unprivileged application performing a privileged task. This can occur either by exploiting a local system vulnerability (*e.g.,* bypassing a permission check), or by asking a privileged application to perform the task on behalf of the unprivileged app (also known as a *confused deputy*[8] attack).

Bugiel *et al.* [24, 25] describe a multi-layered security framework for Android that combines application-level inter-process communication (IPC) inspection, and kernel-level mandatory access control (MAC) to prevent privilege escalation attacks. Their

[8] This attack model is often called a *confused deputy* attack [72], in which an innocent application is fooled into performing a task on behalf of a malicious application. Since the privileged application is allowed to perform the task, and is trusted by the OS, the attack is both difficult to detect and difficult to prevent.

system allows the user or administrator to supply custom policies that can prevent information flows from unprivileged applications to privileged ones. A more lightweight approach was proposed by Dietz *et al.* [39] in which IPC messages on Android are allowed or denied after performing the equivalent of a call stack inspection [122] on Android.

Felt *et al.* [55] proposed a different approach to preventing confused deputy attacks on Android. The authors suggest dynamically reducing the privileges of the callee application to the intersection of the permission sets of the caller and the callee. If this technique is being used, the caller (possibly malicious) application would be forced to request more privileges at install-time, as attempts to use a privileged application without sufficient permissions would be denied.

Chin *et al.* [31] and Kantola *et al.* [78] attempt to identify errors where developers make available APIs to provide functionality to other applications, but do not perform necessary privilege checking to ensure the API call does not leak sensitive data. By allowing *any* other application to invoke API calls, the app making available such functionality may be susceptible to abuse. In the former proposal, manual inspection was done on 20 popular applications, and in the latter a set of heuristics are proposed for automated detection of vulnerable applications.

Ontang *et al.* [97] propose redesigning Android's IPC framework to allow the definition of fine-grained policies on the receiver application. By allowing applications with exposed interfaces to limit which applications (including version numbers, code signing certificates, authors) can use their APIs, the risk of communicating with (and possibly being exploited by) malicious applications may be reduced.

2.5.3 Malware Scanning

Anti-malware vendors often claim that malware on smartphones is rising exponentially and that Android users, due to Android's decentralized model, are at greater risk of infection. One recent paper [82] analyzed Domain Name System (DNS) queries for over 380 million cell phones on a major US cellular carrier, and found that only 0.0009% of devices were observed to be executing malware that communicated with known malicious domains. These findings challenge the notion that malware is ram-

pant on decentralized (and centralized) ecosystems. Two surveys of mobile malware [52, 132] on iOS, Symbian and Android characterize the main malware families (including those found in the study above) comparing infection vectors and common post-infection behaviour.

Proposals for system call analysis for malware detection also exist. Two proposals [26, 102] record system call traces on Android devices, analyzing them locally (*e.g.,* by comparing to a local policy) or optionally submitting them for cloud-based analysis. A different cloud-based approach is Jarabek *et al.*'s [75] ThinAV, a lightweight antivirus for Android that looks up cryptographic hashes of Android applications on freely available cloud-based anti-virus engines.

Zhou *et al.* [131] proposed a method for detecting repackaged[9] Android applications on third-party application markets. Their technique relies on having a known-good copy of Android applications collected from the official Google Play Store. Applications are run through a disassembler, and the resulting code is then run through a *fuzzy hashing* algorithm that is resilient to small changes in the binary. The resulting hashes are compared and a similarity score is assigned to application pairs. In subsequent work [130], Zhou *et al.* improve the detection technique by adding a *module decoupling* stage, where the application is split into its main components (*e.g.,* ad libraries, tracking libraries, static resources, *etc.*). Module decoupling is done by statically computing a program dependency graph on compiled code.

Google themselves reportedly scan for malware applications submitted to the official Google Play store [64]. Google appears to run applications in a virtual Android environment for 5 minutes while monitoring network access, as well as access to file system and other private resources [77].

2.5.4 Kill Switches

Remote application revocation and uninstallation, also known as a *kill switch*, is a powerful feature in centralized operating system ecosystems. Kill switches allow the OS vendor to remotely (potentially without user interaction or approval) uninstall or

[9] Repackaged applications are benign applications onto which malicious code has been attached. The new binary is subsequently published on to a third-party application market.

disable an application on a user device. Kill switches typically work closely with a platform's application marketplace since their operation requires that vendors know (or have some mechanism to detect) if an application has been installed on a device. On-device software to access the vendor's marketplace might periodically transmit a list of installed applications to the vendor. While kill switches offer a mechanism to control the spread of malicious applications, they can also facilitate excessive vendor control and might potentially be abused for censorship or anti-competitive practices.

Notorious examples of kill switches used in practice include Amazon's removal of George Orwell's 1984 e-book from user's Kindle e-readers [4] after the book's publisher noticed that copies had been sold in unauthorized regions. Google has also invoked their kill switch to remove a malicious application from all user devices [100]. Apple has also confirmed that iOS includes kill switch functionality [114], but there is to our knowledge no existing evidence that it has been used in practice.

2.6 Secure Software Distribution and Updates

To the best of our knowledge, there is little academic research related to understanding, defining, and securing the software installation process. This may be unsurprising, since the notion of software installation is relatively recent (circa mid-1990s, as OSs and software grew in complexity, requiring an installation prior to running), and regular updates becoming an even more recent practice. The few academic papers that have been published are discussed in this section and its subsections.

A survey by Anderson *et al.* [5] presented a comprehensive stakeholder analysis of software installation. The work identified key roles in the software installation process such as users, administrators, developers, packagers, certifiers, *etc.* The goals, incentives and influence is discussed for each role, concluding that traditional marketplace phenomena (*e.g.,* asymmetric information and moral hazard) also occur in the software installation ecosystem. The paper also surveyed software installation frameworks used in 10 different platforms (Microsoft Windows, Mac OS, Ubuntu Linux, Symbian, iOS, Android, browser plugins, browser extensions, Facebook applications and OpenSocial), noting that newer OSs targeted towards smartphones and tablets have a more robust security architecture than desktop OSs. At the same

time, browser extensions and plugins were noted as being either highly privileged (*e.g.,* Mozilla Firefox) or poorly vetted (*e.g.,* Google Chrome).

2.6.1 Secure Software Distribution

When developers began using the internet for software distribution, malicious developers leveraged the internet's inherent anonymity properties to distribute malware. Executing code from untrusted sources, sometimes as a privileged (*i.e.,* root) user, can lead to the compromise of the system. The problem for users became how to know ahead of install-time, whether or not the source of the software could be trusted.

Rubin [101] proposed one solution to the problem, which involved a trusted third party (T) issuing certificates for developers (D). In this scheme, D registers with T by proving his identity and supplying a public key. For each software release, D constructs a message m which includes the developer's name, file name and location, version number, cryptographic hash of the file, and a timestamp. The message is digitally signed and submitted to T. Upon receiving m and verifying D's signature, T creates a certificate which includes all information above as well as T's identity. D can now post a copy of his software along with the certificate issued by T. If the user has a copy of T's public key, it is possible to independently verify that the downloaded software was certified by T.

The solution above is the foundation for many trusted software distribution schemes where a central authority exists. The scheme is lightweight for the user who only needs the trusted party's public key and the ability to verify signatures. For example, Apple (centralized) signs and encrypts third-party applications on its app store, and preloads verifying keys onto user devices [42]. Decentralized environments by definition do not involve a trusted central party, so while digital signatures are used, they typically provide a weaker form of authentication [16] as they can only be used for *recognition* (see Trust on First Use; Section 6.2.3).

Many popular package managers[10] have been found to either use signatures incorrectly

[10] A package manager is a software tool that retrieves and installs software from the OSs official servers or from mirrors hosting copies of the software. The package manager may also be responsible for keeping records of installed software and versions, and identifying and resolving dependencies or conflicts between installed packages.

or not at all, exposing users of the package managers to vulnerabilities [105, 28]. In fact, some package managers appear to trust any file, even when obtained over an unsecured channel (*e.g.,* HTTP without SSL or unencrypted FTP). These vulnerabilities allow malicious parties to deliver (through a man-in-the-middle attack) compromised software to users, and possibly execute the malicious code on user devices. Subsequent to the work of Cappos *et al.* [28], many Linux distributions have updated their package managers to verify digital signatures or, at the very least, verify SSL certificates presented by servers.

2.6.2 Secure Software Updates

Bellissimo *et al.* [23] surveyed software update frameworks used by popular OSs and applications (*e.g.,* Windows and Apple update, Firefox, McAffee VirusScan) and found that some are vulnerable to man-in-the-middle attacks. Frameworks are usually at risk if they do not authenticate the binaries that are being downloaded, or the channel through which they are retrieved. The authors note that custom or standalone software update solutions are more likely to be insecure due to developer error, while OS update mechanisms tend to use multiple layers of security (*e.g.,* signed binaries delivered over SSL).

In the simple case where a single private key is used to sign all updates, the compromise of that key can be devastating. To address this problem, Samuel *et al.* [106] propose splitting the update process into several roles. Developers in each role are tasked with different responsibilities (*e.g.,* timestamping the update or packaging files), each requiring a digital signature created with a separate private key. Thus, if a single key gets compromised, users are not immediately at risk of being sent malicious updates. The update framework attempts to divide trust of private keys amongst multiple parties to reduce the risk of a single compromise. This idea is also the basis of *threshold signatures* [109], which may be used as a lower level cryptographic primitive when creating keypairs for the framework above.

Van Oorschot and Wurster [117] discuss using digital signatures to prevent the unauthorized modification of binaries on Linux systems. The general approach presented uses file system metadata blocks to store verification keys for subsequent updates.

If a *locked* file needs to be replaced, the replacement file must be digitally signed such that the signature can be verified with at least one of the on-disk verification keys. Updated versions may include different sets of verification keys, allowing key *evolution* or *continuity*. We build upon type of key continuity scheme in Chapter 6.

2.7 Summary

This chapter has introduced the definition of software installation that will be used throughout this thesis. We've presented a classification of software installation models, comparing properties of centralized, decentralized and hybrid ecosystems. This chapter also covered related work in software security occurring prior to or post software installation. Finally we highlighted relevant related academic work in securing software distribution and updates.

Chapter 3

Android's Software Installation Framework and Security Model

3.1 Introduction

In this chapter, we conduct an empirically-informed systematic analysis of Android's software installation framework, focusing on the security mechanisms employed during app installation. To understand and address limitations of the Android software installation framework, we begin this chapter by providing an in-depth analysis of Android's security model and design.

3.2 Android Security Model

Android inherits many security properties from Linux due to executing the Android runtime and libraries on top of a Linux kernel (see Figure 3.1); every application, including those running in a lightweight Java-like virtual machine (Dalvik[1] [8]), is

[1]On recent Android builds, Google has included a new runtime designed to replace Dalvik. The new runtime is called *Android Runtime* (ART). We note that while documentation for ART is, as of writing, not available, it appears ART is a transparent drop-in replacement for Dalvik, requiring no change for most applications. ART focuses on increasing application performance and lowering power consumption, but makes no changes to Android's security architecture.

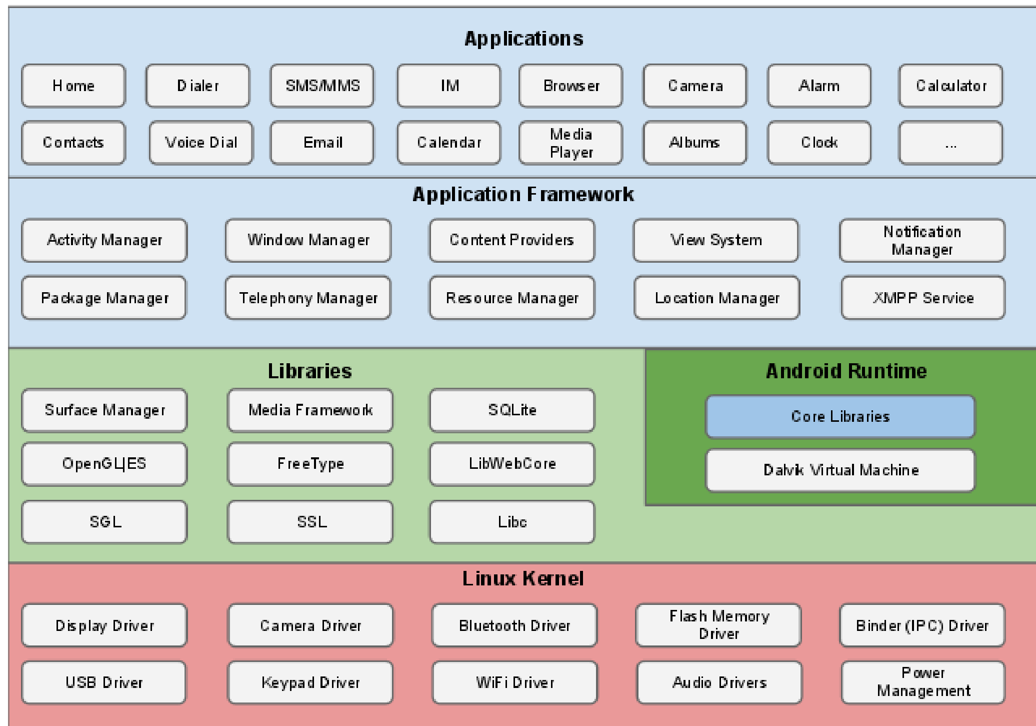


Figure 3.1: Overview of the Android software stack [67].

ultimately a standard Linux process.

A practical example of the relationship between Linux and Android security, and the basis of Android's security architecture is the *application sandbox* [67]. The application sandbox isolates applications and their data through extensive use of a discretionary access control policy (DAC [107]) which by default allows only the application to read and write to the file system location in which it is installed. Leveraging DAC means that every Android application must be assigned a distinct Linux user ID (UID) such that the file system ownership can distinguish between apps[2]. The kernel is then responsible for enforcing isolation between sandboxes, preventing apps from interfering with each other.

By definition, the DAC model allows developers to set access permissions on files their application owns and creates. While it is generally advised to use the default permissions (*i.e.*, read and write only by the owner of the file), it is not uncommon for developers to accidentally or intentionally create world-readable files [93]. As of Android

[2]We note that having each application run under a separate UID is significantly different from a typical Linux desktop, where applications run with the privileges of the user who launched the application.

4.2 (API level 17), creating files as world readable by using the `FileOutputStream` API is deprecated, as this practice has been identified as the cause of many security vulnerabilities. The code snippet below demonstrates how a file can be created with world readable permissions.

```
...
String filename = "myfile";
String string = "Hello world!";
FileOutputStream oStream;

  try
  {
    //can't figure out why there's an error, but this fixes it
    //commenting out for now, return to private before release!
    //oStream = openFileOutput(filename, Context.MODE_PRIVATE);
    oStream = openFileOutput(filename, Context.MODE_WORLD_READABLE);
    oStream.write(string.getBytes());
    oStream.close();
  }
...
```

Figure 3.2: Example Java code for writing to a file within an Android application. The author of this code has made the file world readable to debug a problem.

In order to further protect and isolate applications despite the possibility of developer error, Google introduced mandatory access control (MAC) through SELinux [110] in Android 4.2. The default SELinux security policy provides an additional layer of security on top of the Linux DAC by preventing applications from accessing files outside their sandbox, even if the files are world readable. While SELinux was set to *permissive* mode[3] in earlier versions of Android, as of version 4.4 it is set to *enforcing* [7].

3.3 Application Packages

Android applications are distributed as compressed archives, typically with a `.apk` file extension (leading to the colloquial term *apk file* to refer to an Android application). As shown in Figure 3.3, each application archive contains a Dalvik executable

[3] SELinux can be configured to one of: off, permissive or enforcing. In permissive mode, actions violating the security policy will be detected and logged, but allowed. Enforcing mode will block actions that violate the security policy. Off disables SELinux enforcement completely.

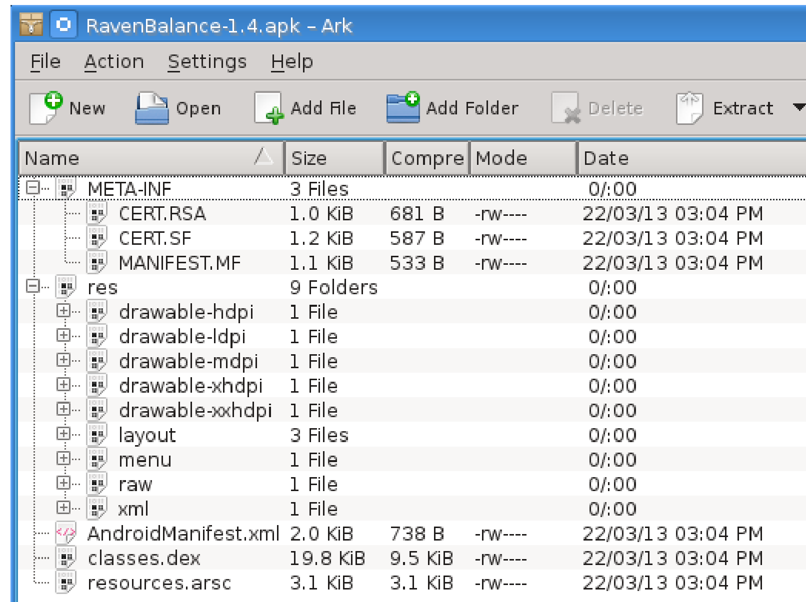


Figure 3.3: Directory and file structure of an Android application archive.

(`classes.dex`), which is the application's main binary and designed to be executed inside the Dalvik VM. The compressed archive also contains a set of resources in the `res` directory (non-executable files such as graphics, media files, user interface components, *etc.*).

Application developers must include a manifest (`AndroidManifest.xml`), which is an XML-encoded text file describing the application's properties. The manifest is read by the system at install-time to display to the user strings such as the app package name, version number, supported Android versions (also known as target software development kit (SDK)), and other attributes. The manifest file also contains a `version code` (*e.g.,* 375 or 12668442), which is a developer-chosen monotonically increasing integer independent of the user-visible `version name` (*e.g.,* v1 or v2.5). During application updates, the OS compares version codes and only allows installation if the version code is set to increase (*i.e.,* app downgrades are never allowed).

Every installed Android app must have a unique, developer-chosen *package name* defined in the manifest, and this name should follow standard Java naming conventions to avoid collisions amongst applications. A general practice is for developers to reverse their web domain name for uniqueness (possibly appending a name for an app if there is more than one app per domain, *e.g.,* `org.mozilla.firefox` and `org.mozilla.firefox_beta`). The package naming convention may be enforced by

application markets such as Google Play, but developers are free to claim their package namespace, or re-use an existing namespace. The lack of namespace control can create confusion amongst users as well as introduce security vulnerabilities as discussed in Chapter 4.

3.4 Permission Model

The Android security model uses a permission-based system that by default denies access to features or functionality that could negatively impact the user experience, the system, or other applications installed on the device. Examples of these features are sending messages or making phone calls, which may incur monetary cost to the user; keeping the device screen on or accessing the vibrator, which could result in battery drain; and reading the user's address book which could result in privacy violations.

To make use of the restricted functionality (which could potentially be dangerous if used in combination with other features, or in a different way than intended by the Android OS designers), Android requires application developers to declare in the application's manifest which of the restricted features are intended to be used by their application. Failure to declare a particular permission will result in the related system call or inter-process communication being denied by the system.

As of December 2013, there are 145 items of functionality which are identified as requiring an explicit permission in order for Android to grant access [69] (a complete list of all Google defined permissions is provided in Appendix B). These permissions control access to network and GPS functions, personal information, system hardware and settings, and many other device features. However, Android is designed such that any third party application can define new functionality (*e.g.,* through an API) and make that specific functionality available to other applications based on *developer-defined* permissions. In the case of developer-defined permissions, Android enforces that both the caller and callee applications have matching permissions (*i.e.,* the callee defined the permissions using the `permission` tag in its manifest, and the caller requested the permissions using `uses-permission`) to allow the IPC to take place.

Permissions are declared in the manifest using the `uses-permission` XML tag fol-

lowed by a common namespace (usually `android.permission.*` for Google defined permissions). Self-declared permissions which other applications can request are labelled with the `permission` tag. The Android manifest contains entries which can be automatically generated by the developer environment but some fields, specifically those related to permission declarations, must be manually entered. Manual permission declaration can lead to application developers over-declaring [51] or erroneously declaring permissions that do not exist [21].

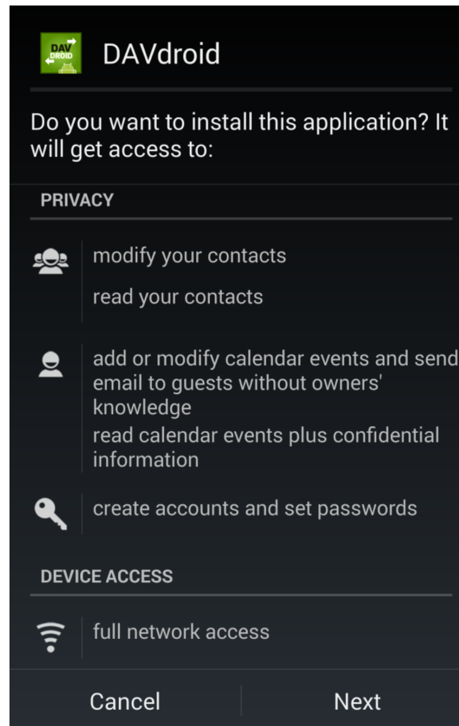


Figure 3.4: Screenshot of permissions requested at install time by an Android application. The install screen displays six permissions grouped into two categories. The user must accept all permissions to proceed with the installation process.

Permissions are enforced by Android at runtime, but must be accepted by the user at install time (see Figure 3.4). When users install a new application in Android (regardless of how the application was obtained), they are prompted to accept or deny the permissions requested by the application. Permissions are also described in a more user friendly (albeit sometimes equally puzzling) language at install time. For example, in Figure 3.4, the developer included the `WRITE_CALENDAR` permission in the application's manifest. This permission is displayed as *Add or modify calendar events and send email to guests without owners' knowledge* on the installation screen.

These descriptions attempt to give a brief, technical explanation of the permission, but do not disclose what the developer intends to use access to those resources for. On stock Android builds, permissions cannot be individually rejected. Thus, developers assume that once their app has been installed they can make use of all the features protected by the permissions requested.

3.5 Digital Signatures

Android applications must be digitally signed prior to being installed on devices [61]. The application installer will refuse to install applications that do not contain a suitable signing certificate. During application development and testing, the standard development environment automatically creates self-signed debug certificates which enables deployment to emulators and devices. For application release, Android allows developers to independently generate or obtain a key pair and a corresponding certificate that can be used for signing apps. The certificate may be self-signed, or issued by a certificate authority. While self-signed certificates are not required, they appear to be implicitly encouraged by Google, since the Play Store only allows apps with certificates expiring after 2033 to be published and many CAs won't issue certificates with a 20 year lifespan.

Developers sign their code through the `jarsigner` [94, Ch. 12] tool, distributed as part of the standard Java development environment. During the signing process, `jarsigner` accepts an unsigned compressed file, and creates the `META-INF` directory inside it (see Figure 3.3). Next, the following three files are added:

1. `MANIFEST.MF` - A text file which contains a list of every file (excluding files in `META-INF`) in the archive at the time of signing. For each file, a corresponding SHA1 hash is stored.

2. `CERT.SF` - A file containing a SHA1 hash of each entry in `MANIFEST.MF` file, along with a corresponding file path.

3. `CERT.RSA` - The developer's X.509 certificate, usually self-signed using the RSA algorithm. This file also includes the signature of the entire `CERT.SF` file.

Apps can be signed with multiple keys, in which case the `META-INF` directory is

populated with multiple certificates (one per signing key), manifests, and signature files. The signature(s) on an app can be stripped by simply deleting the `META-INF` directory.

Digital signatures are used in conjunction with the application sandbox to provide several fundamental aspects of the Android security model. The developer's certificate is used to restrict who can issue software updates to the app (described in detail in Chapter 6), the app's inter-process communication (IPC) abilities, and whether certain permissions are granted. As an example of the latter, the `MASTER_CLEAR` permission allows an app to delete all user data and restore the device to its factory state. While the `MASTER_CLEAR` permission can be requested by any app, it can only be granted to apps signed with the same key as the one used to sign the system image. These types of permissions are called `signature` or `signatureOrSystem` permissions and are further discussed in Section 3.4. Third-party developers may additionally write public APIs and protect them with signature-level permissions.

3.5.1 Certificate Sets

```
int checkSignaturesLP(Signature[] s1, Signature[] s2) {
  ...
  final int N1 = s1.length;
  final int N2 = s2.length;
  for (int i=0; i<N1; i++) {
    boolean match = false;
    for (int j=0; j<N2; j++) {
      if (s1[i].equals(s2[j])) {
        match = true; break;
      }
    }
    if (!match)
    return PackageManager.SIGNATURE_NO_MATCH;
  }
  return PackageManager.SIGNATURE_MATCH;
}
```

Figure 3.5: Android OS signature checking method used in versions 2.2 and earlier.

```
int checkSignaturesLP(Signature[] s1, Signature[] s2)
  ...
  HashSet<Signature> set1 = new HashSet<Signature>();
  for (Signature sig : s1)
    set1.add(sig);
  HashSet<Signature> set2 = new HashSet<Signature>();
  for (Signature sig : s2)
    set2.add(sig);
  // Make sure s2 contains all signatures in s1.
  if (set1.equals(set2))
    return PackageManager.SIGNATURE_MATCH;
  return PackageManager.SIGNATURE_NO_MATCH;
}
```

Figure 3.6: Android OS signature checking method used in versions 2.3 and higher (current).

Android applications can be signed with multiple private keys, generating additional signatures inside the `META-INF` directory. In stock Android, before applying application updates, the following certificate comparison is performed (see Figures 3.5 and

3.6). On current releases of Android (2.3 and higher), the installer builds two Java `HashSet`s from the available certificates[4] in the currently installed app (s1) and the app being installed (s2). If the sets match (*i.e.,* all elements in the one set are identically present in the other), the update is allowed, otherwise the update fails. The code in Figure 3.6 (found within the Android source tree) prohibits developers from ever changing either the number of signatures (and corresponding certificates) in apps, or the certificates themselves.

In previous versions of Android (2.2 and earlier), it was possible to shrink the certificate set as long as all certificates in the old app version were also present in the new (updated) version. As seen in Figure 3.5, this functionality (intentional or otherwise) was possible by iterating through the certificate set in the new app and looking for matches in the currently installed app. Ensuring that N1 is equal to N2 before entering the loop on line 5 would make the two code snippets functionally equivalent.

Additional details on Android's use of digital signatures, including statistics of signing key usage on real-world markets, can be found in Chapter 6.

3.6 Secure application interaction

Android allows developers to use signature information to authenticate applications. This extra level of authentication (*i.e.,* beyond simply looking up the package name of the caller on an IPC message) allows developers to be assured that a message is coming from an application signed with a the same certificate. There are two ways in which developers can make applications interact securely.

3.6.1 UID sharing

As described in Section 3.2, Android applications are assigned unique UIDs at install-time to enforce application isolation and sandboxing. When two or more apps are signed with the same certificate and simultaneously installed on a device, developers

[4] Although the source code references "signatures", signature objects are simply DER-encoded byte representations of X.509 certificates.

can specify that a subset of these apps should be assigned the same UID. By having multiple applications run under the same UID, the applications become part of the same sandbox and are able to share file system access as well as process and memory space.

UID sharing is requested by developers by specifying the `sharedUserId="string"` directive in the manifest of all apps that should share a UID. The signing certificate, as well as `string` label must be the same on all applications sharing a UID. Developers can optionally specify the `sharedUserLabel="string"` as a user-friendly string to describe the set of applications sharing a UID.

Shared UIDs are common amongst modular applications such as browsers and plug-ins. One example of UID sharing found on the Android Observatory (see Section 5) is the Dolphin Browser (`sharedUserId="mgeek.dolphin.browser"`), which shares a UID with extensions such as ad blockers, PDF viewers and translation services. Sharing UIDs allows for tighter integration between applications, eliminating the need the need to use excessive inter-process communication to transfer data. However, increasing the number of apps in a sandbox also increases attack surface and may allow unprivileged applications to perform privileged operations through other applications in the shared UID group (see Chapter 7).

3.6.2 Signature permissions

Android allows developers to define public interfaces and make those interfaces available to other apps via IPC. Some interfaces may be sensitive so developers may wish to only allow known applications to invoke them. To do so, developers can make use of developer-defined permissions. One type of developer-defined permissions is a signature permission, which is only be granted to applications signed with the same key as the application exposing the interface. Developers often use signature permissions to securely expose functionality and interact with other apps.

Defining a signature permission is done by specifying the `permission` directive in the application's manifest, as well as `protectionLevel="signature"`. System applications may specify `protection Level="signatureOrSystem"` to allow applications signed with the same certificate *and* applications installed on the read-only system

partition to request the permission. If an application requests a signature-level permission but does not meet the specified criteria, the application will be installed but not granted the specific permission.

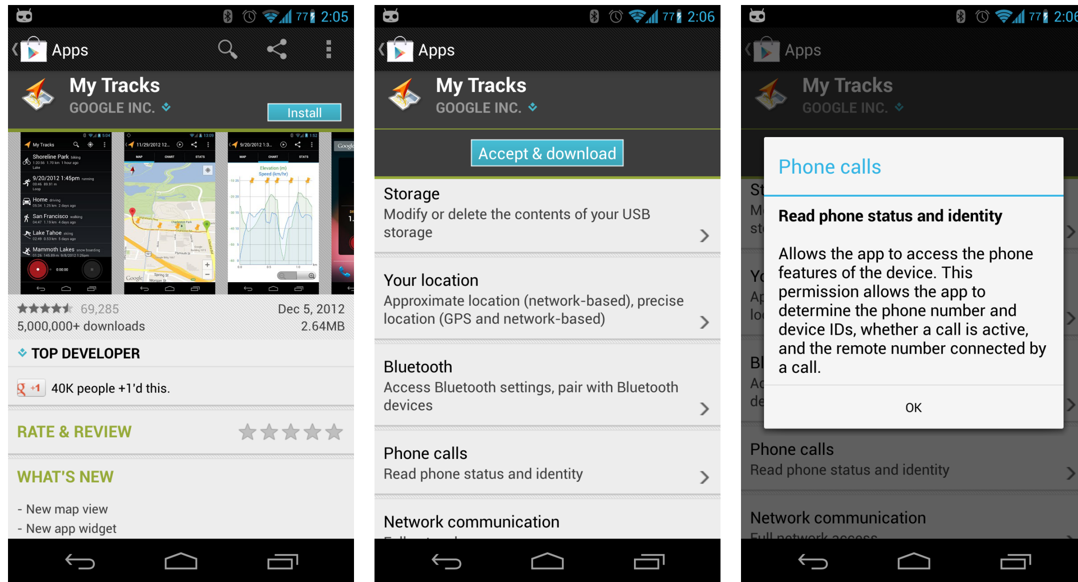
3.7 Installation, Updates and Removal

Android devices are sold in one of two main configurations, depending on the manufacturer licensing agreement with Google. If the device manufacturer has licensed the use of Google services for Android, the Android build will typically include proprietary Google applications such as GMail, Maps, and the Google Play store. Use of these services requires a Google account.

If the manufacturer does not obtain a license for Google services, the device will ship without the inclusion of the Play store and other Google-branded applications. The absence of the Play store requires that users obtain applications through third-party application markets such as the Amazon App store or F-Droid (other popular markets are listed in Section 5.3). As of late 2013, and independent consulting firm [1] estimated that 32% Android devices (accounting for 25% of all smartphone devices) ship with a version of Android that does not include the Google Play store.

3.7.1 Installation with the Google Centralized Store

A typical Android app installation from the official Google Play store requires some level of user involvement as show in Figure 3.7. First, users are given the opportunity to browse through the application's description and reviews (Figure 3.7a). Additional metadata such as number of downloads, date of the latest build, file size, and a changelog is also shown at this step. Research has shown that users infrequently perform detailed inspection of this screen [30]. Next, users are shown a list of permissions requested by the application (Figure 3.7b) and are allowed to optionally view a more detailed description of a permission by tapping on the permission. We note that the detailed permission view (Figure 3.7c) displays pre-defined text rather than text created by the application's developer. Thus, there is no way for the developer



(a) Step 1: View application statistics and developer information.

(b) Step 2: Review list of requested permissions.

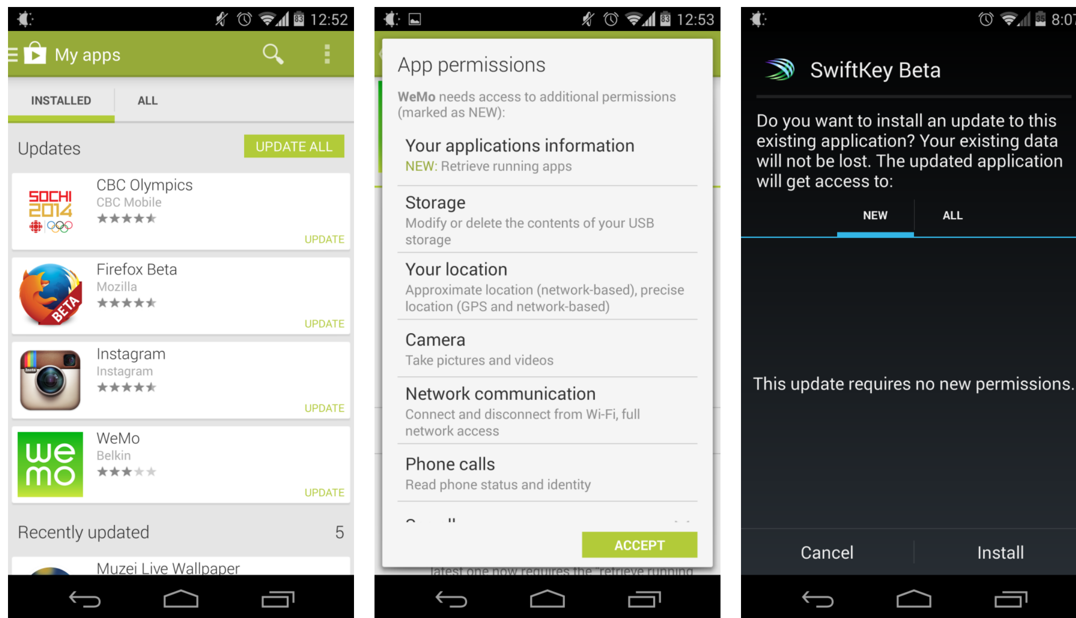
(c) Step 3: (Optional) Read more information about a specific permission.

Figure 3.7: Screenshots taken during app installation through the official Google Play store

to explain or justify the use of a particular permission. Research has also shown that users do not inspect detailed permission listings, nor do they understand what the permissions are displaying [54].

3.7.2 Installation without the Google Centralized Store (Sideloading)

If the Google Play store is not available on the device, users can obtain applications by downloading the application packages directly (*e.g.*, via a web browser). Directly installing applications is sometimes referred to as *sideloading*. Third-party application stores often offer an on-device client (which must itself be sideloaded) to interface with the server component of the store. Clients may mimic some of the Google Play store user interface (*e.g.*, displaying number of downloads or ratings), but they are only responsible for handling the retrieval of the application package, and invoking the system installer.



(a) Google Play: Notification of available updates.

(b) Google Play: Review list of requested permissions. If new permissions are requested, they are displayed as NEW.

(c) Sideload: New and existing permissions are displayed in separate sections.

Figure 3.8: Screenshots taken during app updates through the Google Play store ((a) and (b)) and sideloading (c).

If the application package was manually copied on to the device, a file browser must be used to open the package. Android's installation framework notices that an `apk` file is being directly opened, and will present the user with a screen similar to Figure 3.4. If the user accepts the permission listings, installation proceeds.

3.7.3 Updates

The Google Play client can notify users when updates for applications are available (see Figure 3.8a). Google achieves this functionality by maintaining a list of applications and their versions per user and device. When a developer submits an updated version of an application, Google can notify all users of the outdated version that an update is available. Users may also opt-in to automatic (silent) updates on a per-app basis.

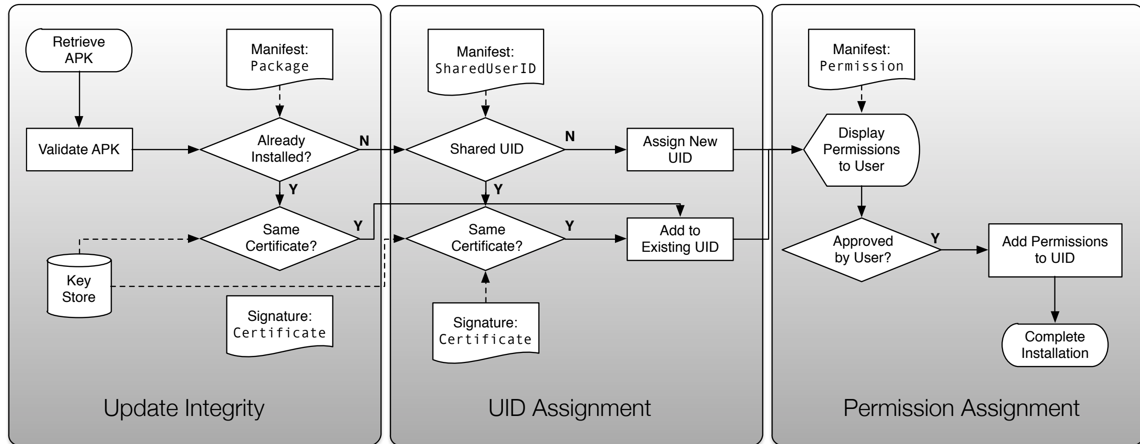


Figure 3.9: Our abstract model of the Android installation process for an app package. Unless otherwise specified, an answer of no to any conditional aborts the installation.

When accepting an update, the permissions requested by the application may have changed (*e.g.,* in the event of a new feature being added). If permissions change, users are shown the full permission list, and the new permissions are shown at the top of the screen (see Figure 3.8b).

When applications are sideloaded, the user is responsible for noticing and downloading updates unless a third-party market client provides that functionality. Sideloading an update displays a screen as shown in Figure 3.8c, where new and previously accepted permissions are shown to the user.

3.7.4 System workflow

When a new Android app is loaded for installation, either through a third-party market or side-loading, Android's software installation framework follows the sequence of events as depicted in Figure 3.9. We note that when apps are installed through the official Google Play Store, permissions are approved prior to installation (see Section 3.7.1), but the rest of the process remains the same.

First, the app package validity is verified: the system ensures that the Android app package is indeed a compressed archive, has not been modified or corrupted since being signed, and that it contains a valid certificate for the signing key. Next, the installation framework decides whether the app is a new installation or should overwrite (*i.e.,*

update) an existing app. If the app being installed has the same `package` attribute in the manifest (*e.g.,* `com.google.android.music`) as another currently installed app, then Android will treat the installation as an update.

In the update scenario, the certificate (or set of certificates if signed by multiple keys) of the new app is compared to the certificate(s) of the already installed app. If both apps were signed with the same key(s), then the currently installed binary is removed (preserving any user data) and the new app is installed in its place. Otherwise, the new app is installed as an initial installation. We note that it is the certificates themselves that are compared, not the signatures. Thus, even if two certificates are signed with the same private key, updates are not allowed. Update integrity is covered in detail in Chapter 6.

Next, Android must assign a UID to the app. If an update is taking place, the previously assigned app's UID is preserved. In the case of an initial installation, Android checks if the app's manifest contains the `sharedUserId` directive. If so, the system looks for any other installed apps that are signed with the same key(s) and also have `sharedUserId` specified in their manifest. If such apps are found, the app is assigned the same UID; otherwise a new UID is created.

Finally permissions must be assigned to the UID. The user is prompted to review and approve the permission assignments before the app is installed. When UID sharing is not used, permissions listed in the app's manifest are assigned to the UID. When UID sharing is used, the UID is assigned the union of all permissions in the manifests of apps sharing the UID. If the app is updating an already installed app, the permissions listed in the updated app's manifest are assigned to the UID. In Chapter 7, we describe limitations with Android's current UID sharing method and present a more flexible scheme. We also describe the *permission inheritance* attack, through which an unprivileged application is granted additional privileges.

Uninstalling apps

Applications on Android cannot uninstall other applications without user interaction. Uninstalling an app typically requires the user to load the on-device application market or application manager. Uninstalling an app removes all locally stored user data, but applications may store data elsewhere, (*e.g.,* on a remote server or an SD card).

The method in which that data is handled after uninstallation is up to the developer.

Downgrades

On Android, application downgrades (*i.e.,* replacing a newer version of an application with an older copy) are not allowed. While the motivations for this are not explicitly stated, we assume that allowing downgrades may allow an attacker from posting a correctly signed, earlier, vulnerable version of an application and make it available for download. Downgrades are prevented through use of the `version code` (see Section 3.3), which must always be greater than the currently installed code, thus acting as a type of timestamp on Android applications.

In practice, other environments where applications self-signing is used typically discourage downgrades for the same reasons stated above [106].

3.8 Summary

This Chapter has provided an in-depth description of the Android security model and explained the inner workings of its software installation framework. We discussed basics about Android applications and permission model. We covered Android's use of digital signatures and their relation to the overall security model. Finally, we discussed how application installation and updates are done from the perspective of both users and the system. Many of these concepts will be revisited in the following chapters when discussing and addressing limitations of Android's installation framework and decentralized software installation in general.

Chapter 4

Securing Software Discovery

4.1 Introduction

In this chapter, we focus on the security of the initial step in the software installation process: software discovery. During this step, users first decide that a new piece of software is required to accomplish a specific task, and proceed to search for an application that suits their requirements. This step may also involve being notified that there is an update available for an application that is already installed. In a centralized environment, the process of discovering new software is constrained by the applications made available by the central party hosting, for example, a centralized application store. In the absence of a central party, users will typically have more options available for finding software, but may have difficulties identifying benign applications.

Decentralized environments allow the existence of multiple application markets, allowing users and developers to consume and create software freely. In addition, modern operating systems such as Google's Android or Apple's iOS have created software installation interfaces which enable users to complete the software installation process with only a single click or tap (see Section 3.7.1). This chapter examines the security implications resulting from the convergence of one-click installation in decentralized environments.

Specifically, we first identify security properties that are lost by allowing the existence

of multiple application markets. We propose a software installation framework with the objective of achieving single market security semantics while retaining the flexibility and independence provided by a decentralized setting. The proposed solution described in this chapter (which we call Meteor) enhances app installation with an extensible set of configurable security *information sources* and *kill switch authorities*. At install time, information sources provide the user with additional app information, ranging from app age (*i.e.,* time since the app's first appearance), virus reports and privacy violations, to expert ratings on the app and other apps by the developer. Kill switch authorities allow consumer devices to be configured to subscribe to notifications of dangerous apps for removal.

A fundamental goal of our approach is to connect digital properties such as package signatures to human evaluable information such as developer and application names. Our key observations are that (1) name collisions in both developer and application names should be minimal even in a decentralized ecosystem; and (2) name collisions should in fact raise suspicion, since the most frequent cause of a name collision is malicious substitution of an app [30].

4.2 Background

We first briefly discuss the benefits of centralized ecosystems as well as the motivation for decentralization. Next, we discuss the role of digital signatures on smartphones. The section concludes with a review of current malware mitigation techniques on smartphone platforms.

Note that this and the following sections frequently discuss properties specific to the Android platform. We do this for two reasons. First, Android is the first platform to experience the negative effects of a multiple-market ecosystem. Second, focusing on a specific technology simplifies explanations. However, the architectural lessons are generally applicable across platforms.

4.2.1 Application Markets

As discussed in Section 2.3.1, a single, central application market offers an opportunity to improve overall platform security. To date, this model has exhibited several clear advantages. First, a market acts as a centralized choke-point for detection of malware[1] (*e.g.,* Google's Bouncer [64]). Second, it provides a means to remotely uninstall distributed apps later identified as malicious.

These remote uninstalls (or Kill Switches; see Section 2.5.4) provide faster clean-up than traditional antivirus software, as they push actions to devices and do not require client-side definition updates or resource intensive scanning. Finally, search results display consistent developer names for applications[2].

Once a developer has registered with a central market, controls exist such that no other developer can easily distribute applications under that developer name. Hence, consumers have some assurance that all applications provided by "John Smith" are provided by the same John Smith. While this characteristic does not ensure that "John Smith" is the John Smith the consumer intended, it does allow the app market to apply sanity checks for well known, high-impact entities, *e.g.,* "Bank of America."

Motivation for multiple markets

Each platform usually provides its own official application market. However, strict terms of service and questionable motivations for acceptance of an app in a specific market [49] often motivate multiple markets. For example, the Cydia market has been created for "jailbroken" iPhones and the Amazon Appstore has been created to provide developers with better marketing capabilities. Multiple application markets present a different use model to the consumer. To use multiple markets, the user must currently download and install separate applications which serve as gateways into each market. This process may in fact negatively impact the security of the device (*e.g.,* iPhones must be jailbroken to install Cydia; the Amazon Appstore requires users to disable the security feature that disallows applications from unknown sources). Next,

[1] We note that while market-level app vetting can be useful under some circumstances, the practical effectiveness and scalability of this approach remains uncertain [86].

[2] When corporations commission developers to create smartphone applications the corporation name may be listed rather than the developer name.

managing apps becomes more difficult. Users may become confused when managing apps available in multiple markets. Finally, if a malicious app is identified, it is unclear which market has the authority, responsibility, and capability to employ a kill switch.

In a multiple-market environment, individual market vendors can compete by offering the same app at a lower price (*e.g.,* the Amazon Appstore has a free "app of the day"). This allows consumers to comparison shop to find the cheapest version of an app. From an economics perspective, such competition is healthy. However, without a mechanism to determine if two or more apps are actually the same app, an adversary can lure users to install a malicious version of an app by distributing it into a market where the legitimate app does not exist. Attackers may also sell the app at a lower price to receive direct monetary revenue in addition to the return provided by the malware.

4.2.2 Package Signing and Application Namespace

Modern smartphone OSs require applications to be digitally signed [22]. On Android, app signatures are implemented as a form of *continuity signing* [117]: the OS verifies that subsequent application updates were signed by same developer as the original.

In order to sign an app, developers generate a public/private key pair and a self-signed certificate. As certificates are self-signed and never displayed to the user, nothing prevents the developer from inserting fake or incorrect information into the certificate. Once an app is ready for distribution, the developer uses the `jarsigner` [94, Ch. 12] tool to sign the app and embed the signature into the application package. Android does not allow adding or removing certificates or keys during app updates [117], which forces developers to release a new application under a new package name (possibly losing user data; see Chapter 6) in the event of losing their private signing key.

Android internally uses the *package name* (*e.g.,* `com.company.app`) as an identifier for installed apps, and a new app is subjected to the continuity verification mentioned above if it is identified by the same package name as a currently installed app. If there is no currently installed app with same package name, the app is installed without additional checks. Continuity signing is verified for that package name from that

point forward.

Package namespace collisions can and do occur. Occasionally, collisions result from careless developers who do not choose a sufficiently distinguished package name. However, package name collisions can also be a sign of an attack. For example, the *Geinimi* trojan [83] grafted SMS-sending malware on to the popular Monkey Jump 2.0 game. The malware was distributed through an alternative app market for Android and used the same package name (`com.dseffects.MonkeyJump2`) as the original game. As a side-effect, the name collision would have prevented a user from later installing the legitimate app without previously uninstalling the malicious version.

4.2.3 Malware Mitigation

Similar to commodity desktop platforms, antivirus software has emerged for smartphones. However, smartphone antivirus software does not operate analogous to desktop antivirus software due to differences in the execution environment. First, energy consumption prohibits routine scanning of files. Second, since smartphone apps run within a sandboxed environment (*e.g.,* as in Android and iOS), antivirus apps do not have sufficient low level API hooks. Instead, smartphone antivirus programs frequently maintain blacklists containing identifiers for malicious applications. The resulting functionality strongly resembles that of kill switches.

The kill switches deployed by official markets are commonly integrated with the OS platform. If malware is detected to have been distributed through the market, the kill switch can remotely uninstall the app from phones. If the phone has network connectivity, the removal can occur nearly instantaneously. For example, Android maintains such a connection for Google services, including app installation and removal. Otherwise, removal occurs when connectivity is restored. An advantage of kill switches over antivirus software is the ability for a remote server to decide which installed apps to remove, as the market already maintains the list of installed apps. This eliminates the need for the phone to download and maintain blacklists, which conserves battery life.

We note that kill switches operate under the assumption that the malicious app was contained by a sandbox. In March 2011, Android malware exploited a local privilege

escalation vulnerability [63], which required more attention than simply uninstalling the malicious app. As hinted above, Android's kill switch mechanism is implemented as part of a protocol that allows both removal and installation of apps. Therefore, the Google Android Market automatically installed a security fix to clean up devices known to have installed the malicious apps. In this case, Google effectively used Android's enhanced kill switch to distribute an OS patch. While kill switches clearly offer valuable functionality, it is less clear what trust consumers should place in each application market.

4.3 Threat Model

The goal of our system is to help users minimize the risk of installing a malicious application. We consider "malicious" applications to be those that contain code to harm user devices, violate user privacy, or perform unwanted actions such as sending SMS without user approval. We classify malicious applications of interest to our architecture into three broad categories depending on the state of installed apps on the device, as well as the package name and signature of the malicious app.

1. **Colliding malicious app:** A malicious application with a package name matching that of another currently installed application. The application signature is different than the currently installed app, so Android would prevent the app from being installed unless the current version is removed by the user.

2. **Non-colliding malicious app:** An application with a package name that is different from any currently installed apps. This category of apps includes look-alikes *e.g.,* Google Maps (`com.goggle.maps`); new apps with original functionality; and weaponized apps with trojaned components. The main characteristic of these apps is that their package names do not collide with the namespace of currently installed apps, therefore triggering no alerts (aside from the standard permission approval) to the user [30].

3. **Rogue update:** A formerly benign application is updated with a properly signed new version that includes malicious functionality. This can happen when a developer's signing key is compromised or the developer him/herself (or part of the development team) goes rogue.

Meteor can help detect malicious applications in categories 2 and 3, but does not detect applications in category 1, since those are blocked from installation by the OS (see Section 4.2.2). We note that intentional user circumvention of this block (*e.g.,* by uninstalling the current version) would effectively move the app to category 2.

Assumptions. We assume the adversary can successfully submit malicious apps to any or all application markets. Adversaries may also independently host their own application repositories. We also assume the adversary is capable of creating duplicate apps that are identical to the original, with the exception of the digital signature. That is, we assume the adversary cannot easily access or independently re-create to the original developer's signing key.

4.4 Meteor: Enhancing the Security of Multi-market Platforms

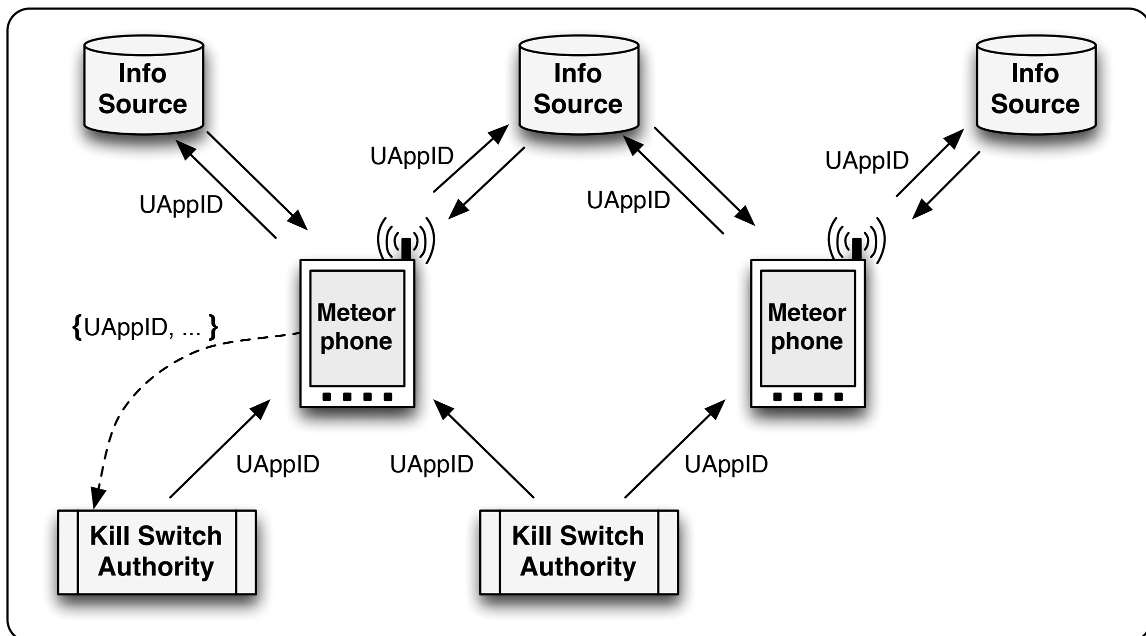


Figure 4.1: Overview of the Meteor architecture. Each device running the Meteor client app (Meteorite) subscribes to a set of information sources and kill switch authorities. Information sources are queried (using UAppIDs) at install-time while kill switch authorities broadcast offending UAppIDs post-installation.

Meteor provides a flexible architecture for addressing the security concerns of a de-

centralized ecosystem. Addressing the app name consistency problem is a significant challenge. For example, simply forcing a fixed namespace based on the Domain Name Service (DNS) would be impractical to authenticate reliably, and it is unrealistic to expect consumers to separate safe from unsafe names. Instead, we leverage universally unique naming, and then aggregate statistics and expert ratings from multiple sources to enable a safer install-time environment.

Figure 4.1 overviews the Meteor architecture. At a high level, Meteor works as follows: after downloading an app, and prior[3] to installation, a pre-configured set of online information sources is queried to obtain additional information about the app or about the app's developer. Each of the sources provides information which may help the user, or client-side app, gain confidence or reconsider the installation of the app. The device may also subscribe to one or more kill switch authorities, which enable the remote uninstallation of apps if they are found to be malicious.

Information sources queried by the Meteor client app (Meteorite) may be grouped by areas of expertise. For example, one source might host information for apps related to social networking while a different source may focus on games. In fact it would be natural to expect multiple sources of each type, used by different users or user groups in different geographic regions. Narrowing the scope of coverage allows experts to use domain-specific knowledge to contribute higher quality information about apps.

The Meteor architecture is based upon the following four fundamental underlying concepts. The remainder of this section discusses these concepts in greater depth.

Universal application identifiers (UAppID): Meteor requires a reliable method of uniquely identifying app instances for reference by both information sources and kill switch authorities. To guarantee uniqueness, identifiers are cryptographically bound to the binary code of app instances, as they incorporate the application name and digital signature data.

Developer registries: The first general type of information source is aimed at developers. Developer registries serve as repositories of developer submitted data such as contact information, news about app updates or issues, and other apps the developer has authored. While a single developer registry is ideal from a security

[3] Optionally, additional checks could be deferred to the background and done after install-time, at the risk that malicious activity could occur in the meantime or alter security enforcement.

perspective, we expect region-specific developer registries to be more amenable to various global political climates.

Application databases: The second general type of information source targets applications. These sources host app information manually entered by experts in a particular domain (*e.g.,* location-based shopping apps), or information automatically gathered from crowdsourced submissions. For example, databases may include expert reviews, ratings, permission descriptions, number of downloads, *etc.*

Kill switch authorities: In Meteor, trust in a remote party to uninstall malicious or inappropriate apps is decoupled from application markets, as markets have varying levels of trustworthiness. Depending on consumer privacy requirements, kill switch authorities may maintain a per-device list of installed apps

4.4.1 Universal Application Identifiers

The Amazon Appstore and the Google Market (and possibly others) use an app's package name to uniquely identify apps inside their stores. In a multi-market environment, using a package name alone will not uniquely identify apps because these names are sometimes arbitrarily chosen and may deliberately collide with another completely independent application (see Section 4.3).

We propose the use of a universal application identifier (UAppID), to provide a common handle for referring to specific application instances. In Meteor, UAppIDs serve two main purposes: 1) *precise app lookups*, useful for comparing apps across multiple markets (similar to the way consumers search for the best price of a specific TV model as opposed to the best price of any TV); and 2) *instance-specific kill switches*, which allow for the removal of specific binaries instead of any app with a given package name.

Using a signed Android app as input, we construct its unique UAppID by extracting the package name and developer certificate (the two components used by Android to ensure correct application continuity as discussed in Section 4.2.2) from the application package. Next we remove the application's signature[4] and certificate, and

[4]Removing the signature of a signed Android app can be done by deleting the META-INF direc-

compute a cryptographic (collision-resistant) hash of the binary. Application names and versions are not explicitly incorporated because they are already embedded within the unsigned binary. Therefore:

$$\text{UAppID} = \{H(\text{package name, dev. cert}), H(\text{binary})\}$$

The UAppID is a tuple of size two. The first element is a hash of the package name and developer certificate[5]. The first element of the tuple can be used to identify all apps that the Android installer considers to be equivalent. The second element is a hash of the application's binary. This portion of the UAppID can help identify repackaged (including pirated, but unmodified) apps across markets, or different version releases of the same app. The UAppID tuple results in a globally unique string that can be used to identify a particular version of an application using a specific package name and written by a specific developer or organization.

This approach has the advantage that UAppIDs can be quickly reconstructed on the device, and provide strong guarantees that two identical UAppIDs refer to the same executable app if a strong hash function is used and verified. Of course, a universally agreed upon collision-resistant hash function must be designated and used (*e.g.,* the Meteorite proof-of-concept app uses SHA256). UAppIDs, similar to other hash functions, can be further encoded to be more human-readable or allow for easier comparisons [99].

4.4.2 Developer Registries

One uncertainty users face when installing applications results from the lack of available information about the app's developer. Installation screens (*e.g.,* when installing via an application market; see Figure 3.7) typically show the developer's name and website, but these two pieces of information are provided by developers themselves, and are generally unverified by the market vendor. Furthermore, apps that are distributed outside of app markets (*e.g.,* through a developer's website[6]) don't generally

tory inside the app package.

[5]Due to the two inputs to this hash function being developer-supplied, we also include the length of each string as input to the hash function to help prevent attacks that rely on the concatenation of variable-length inputs.

[6]Developers might choose to distribute apps on their own to avoid paying registration fees or to keep 100% of the app's revenue.

Type of Information	Description
App binary properties	The name, version, package name and full developer certificates included in the binary.
App age and origin	How long an application has existed and a list of markets or sites on which it has been made available. This can help users determine if an app is brand new, or can be found elsewhere.
Expert reviews (or links to)	This service allows (ideally well-known experts or sets of) users to test software and submit technical or security reviews.
Blacklists and whitelists	The presence of an application (or other applications by the same developer) on a blacklist or whitelist. The reason(s) for being added to a list could also be recorded (*e.g.,* privacy violations, tasteless content, malware, etc.).
Anti-virus and other security and privacy tests	Whether the app has been flagged by an anti-virus tool or has been reported for privacy violations.
Number of installs and uninstalls	Market provided data related to the total number of installations or number of active users of an app identified by a particular UAppID.

Table 4.1: Example types of information that may be provided by application databases.

display *any* developer information at install-time.

We argue for the establishment of one or more central locations where developers can voluntarily disclose more information about their apps, development cycle, company, etc. These registries would be consulted to help resolve the *which John Smith?* problem, or simply to learn more about a developer before installing their software (similar to the way one might research a charity prior to donation). A widely used or relied upon central developer registry would ideally motivate developers to opt-in as a best-practice.

Developer registries can be used in several ways. First, application markets can obtain further confirmation of a developer's history and portfolio during sign-up. Second, application markets can show a "more info" section on an app information screen which hooks into the developer registry to show additional data about the developer. For example, *Registered developer since 04/2011*, *Developer of 3 other apps*, *No apps killed to date*, *4 apps issued to date*, *etc.* Third, they provide a central point for a kill switch authority to look up developer contact information if necessary before deciding to throw a kill switch for their app. Finally, they provide a website that developers can use to make announcements about app issues or updates.

Our goals for such voluntary developer registries do not include becoming a certification authority for developers. These registries are intended to be lookup services

for retrieving information that may help increase confidence in a developer. As such, the enrolment process is envisioned to include only minimal verification of submitted data (*e.g.,* only verifying e-mail retrieval capabilities and control of a signing key), but fundamentally cannot provide assurance about the security or accuracy of provided information. We expect, however, that experts in each developer registry will help identify false or misleading information.

Participating in one or more developer registries should not be made mandatory as this would be both technically challenging and go against (*e.g.,* Android's) open development philosophy. Indeed, if developers wish to remain anonymous, submit false information to the developer registry, or not register at all (*e.g.,* an anti-censorship app developer concerned about political persecution) we envision that their apps would still be installable by those who choose to take that risk. However, we believe that in most cases, positive incentives and benefits would motivate developers to opt-in.

Enrolment: A developer creates a password-protected account on a registry by providing a valid e-mail address. After signing in, the developer asserts ownership of one or more certificates (i.e., app signing keys). The proof of control of the signing key is done using standard known techniques for challenge-response based on digital signatures [87, pp. 404-405], and could be designed to make use of current code-signing infrastructure and tools (see Section 4.2.2).

Hosting registries: Registries would ideally be provided free of direct charge for both developers and users retrieving information. Candidate host providers for this type of service might be universities, non-profits, or commercial organizations funded by advertisement revenue.

4.4.3 Application Databases

Similar to the developer registries above, databases containing information about apps available both within and outside of markets would be valuable to end-users and experts. We see application databases as extensible repositories of application properties, statistics and reviews by subject area experts. For example, cartography enthusiasts can populate a database of apps that relate to maps, while gamers can maintain a database of game apps. We acknowledge that enthusiasts are not always

ID	Name	Version	Package Name	Cert. Hash	App Hash	First Seen	Submissions
1	Chess	1.5.2	`com.games.chess`	0x74A8...	0x93B1...	Jan 2, 2012	100
2	Chess	1.6	`com.games.chess`	0x74A8...	0x8F2C...	Jan 30, 2012	80
3	Chess	1.5.2	`com.games.chess`	0x1D51...	0xA33C...	Feb 21, 2012	3
4	Checkers	0.5-beta	`com.games.checkers`	0x74A8...	0xDB89...	Mar 1, 2012	10
5	Chess	1.6	`com.games.chess`	0xF307...	0x8F2C...	Mar 3, 2012	1

Table 4.2: Example application database. Hashes truncated for space.

security experts. However, they often have a vested interest in the security of apps in their area, so one might expect these enthusiasts to also communicate with security experts, or at least advanced users who report technical anomalies.

An application database contains a new entry for each known instance of an app. Each entry in the database lists information obtained from the package metadata (*e.g.,* app name, app version) and the package name. The entries would also list the hash(es) of the developer certificate(s), as well as a hash of the relevant application binary and any other useful data for evaluating an app. Example types of information are listed in Table 4.1.

As an example, Table 4.2 shows a list of 5 app instances that could have originated from a board-game app database. By viewing and comparing entries, experts managing the database could make the following observations (items with * trigger a warning to the user):

- **App versioning.** (Apps 1 and 2). These apps use the same package name and are signed with the same key. The version numbers and application hashes differ, as expected for a new release.
- **Multiple app developer.** (Apps 1, 2 and 4). A developer in possession of a certificate with hash starting with 0x74A8 is developing the Checkers and Chess apps.
- ***Certificate change.** (Apps 2 and 5). App hashes are the same indicating no code modification, but the certificates do not match. Possible explanations are that the developer lost access to the signing key, the developer is using different certificates for different distribution channels, or an attacker has stripped off the certificate from the binary replacing it with a new one. All these cases trigger further investigation, or notification of suspicious behaviour to the user or an agent working on behalf of the user.

- ***Namespace collision.** (Apps 1 and 3). These apps share a package name and app name, but differ in certificate hash and application hash. This could mean that a malicious developer has grafted malware on to the legitimate chess application, or that by chance, 2 developers have chosen to use the same package and app name, as well as coincided in version number.

In the event of certificate change and namespace collisions, the application database itself cannot answer the question *which of the conflicting apps should I trust?* The information merely suggests the presence of malicious activity, and a supplementary mechanism is necessary to decide which of the apps should be downloaded. For example, the database might also contain initial submission times or overall number of submissions. The database could also include expert reviews, or link to the appropriate developer registry.

We have created an application database called The Android Observatory, available at http://www.androidobservatory.org. This application database contains information about applications crawled from various application markets as well as user-submitted samples. We describe the design and implementation of the Android Observatory in Section 5.

Populating Application Databases: Databases should contain relatively up-to-date information to be effective in helping users make informed decisions. The database could be populated in several ways: (1) by application market vendors relaying information about apps they have accepted into their respective markets; (2) by paid employees who look for apps and submit them to a database; (3) by crowdsourced submissions from volunteers and interested users.

The existence of an app in a database reflects the notion that the app is known or has been seen. The absence of an app in a database could also be leveraged to identify obscure or "fly-by-night" apps. The devices of conservative users may be configured to only allow installation of well-known apps for which no suspicious behaviour has been reported (see Figure 4.5a).

Transparency in Consulting a Database: It is unreasonable to expect typical end-users to actively inspect all information provided for an app. Detailed information retrieval is an option that can be enabled by experts, but ordinary end-users (or agents working on their behalf) would only be alerted in the event of suspicious behaviour

(*e.g.,* certificate change, namespace collision, blacklisted app, *etc.*). Deployed this way, the application database and the client app (Meteorite) would be transparent to users the vast majority of the time as the most common scenario for new app entries in the database would likely be new app versions. Mobile security vendors, security researchers and other automated tools may actively consult the database to find app inconsistencies and overall app statistics, and to compile aggregate statistics.

4.4.4 Kill Switch Authorities

The final component in the Meteor architecture is a kill switch authority responsible for the remote removal of apps on user devices. Kill switch authorities have the unique ability to remove an app from subscribed devices independently of the market or website from which they were obtained. This is different to the way kill switches are handled in current smartphone ecosystems, where each market is responsible for removing apps that it distributed.

Kill switch authorities have the capacity to securely broadcast offending UAppIDs to registered devices at any given time (*i.e., push*). This is in contrast to developer registries and application databases which serve information based on a query (*i.e., pull*). Optionally, devices may transmit lists of installed apps to kill switch authorities (as denoted by the dotted lines in Figures 4.1 and 4.2) such that kill switch signals are only received for installed apps. Informing a kill switch authority of installed apps has significant resource consumption advantages for mobile devices, as the authority can perform management logic and only contact the device when necessary. However, submitting a list of installed applications to a third-party may be viewed as a privacy risk, which may dissuade privacy-conscious users from opting in to this mode of operation.

Similar to application databases and developer registries, we expect kill switch authorities to specialize on a small set of application domains. For example, a kill switch authority could concentrate on identifying and disabling apps which are unsuitable for children younger than a certain age.

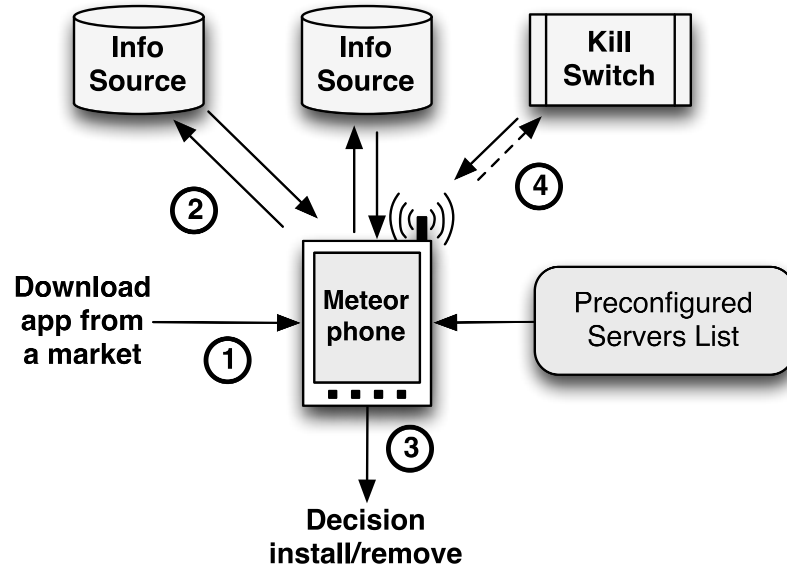


Figure 4.2: Overview of the Meteor app installation process.

4.5 Implementation of the Meteorite Client Application

We have implemented a Meteor client app (called Meteorite) for Android along with the corresponding server-side components to test the technical feasibility of our proposal. Meteorite can be installed on any device running Android 2.2 or greater (which covers approximately 93% of the worldwide Android install-base as of April 2012 [65]. It has three main components as discussed in the following subsections, and requires neither modifications to the underlying OS nor root access. Section 4.5.4 discusses the advantages and limitations of operating within unprivileged app boundaries.

4.5.1 Information Source Management

When the Meteorite app is first installed, the user adds new information sources specifying the type (application database, developer registry or kill switch authority) as well as a URL to query. To minimize the amount of data to be manually entered, servers (regardless of their type) publish a JSON-formatted manifest file that lists services offered and additional server information (see Figure 4.4). A screenshot of the information source entry process is displayed in Figure 4.3.

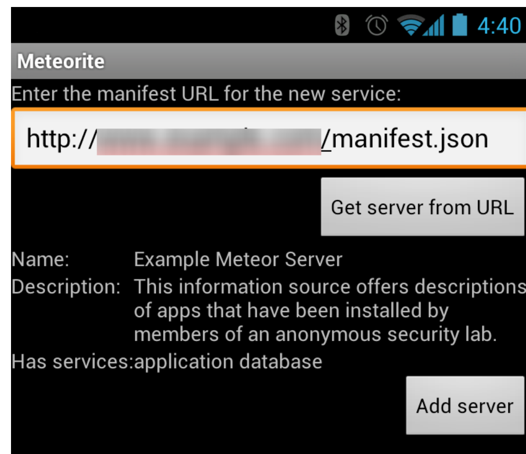


Figure 4.3: Adding an information source to the Meteorite app

```
manifest.json
{ "name" : "Example Meteor Server",
 "description" : "This information source
                 offers descriptions of apps
                 that have been installed by
                 members of a security lab",
 "services" :
 [ { "type" : "application database",
    "url" : "http://XXXXXXX/query.php",
    "license" : "no" } ] }
```

Figure 4.4: The server manifest corresponding to the example in Figure 4.3.

From a practical standpoint, popular or well-known information sources could be pre-configured within the app. We expect users to find other candidate servers via colleagues and expert websites that recommend good sources for given tasks and report poor quality or malicious sources.

The ideal number of configured servers will vary according to security, usability, and cost requirements of each user. Expert users who want as much information as possible with full control over their installed apps may choose to configure a large number of information sources and no kill switch authorities. Non-experts and users who don't want to be involved in low-level technical details of making security decisions may consider installing more kill switch authorities.

4.5.2 Information Source Query

Once information source(s) have been added, the Meteorite app will receive a *Broadcast Intent* every time a new package is installed or updated.[7] Upon receiving the intent, Meteorite computes the newly installed app's UAppID, and issues a standard HTTP POST request to all enabled application databases and developer registries (see item 2 in Figure 4.2). Information sources currently send back information related to the submitted UAppID if it exists. Depending on how Meteorite has been configured, the retrieved app information can be displayed to the user, who is then prompted to continue or to uninstall the app.

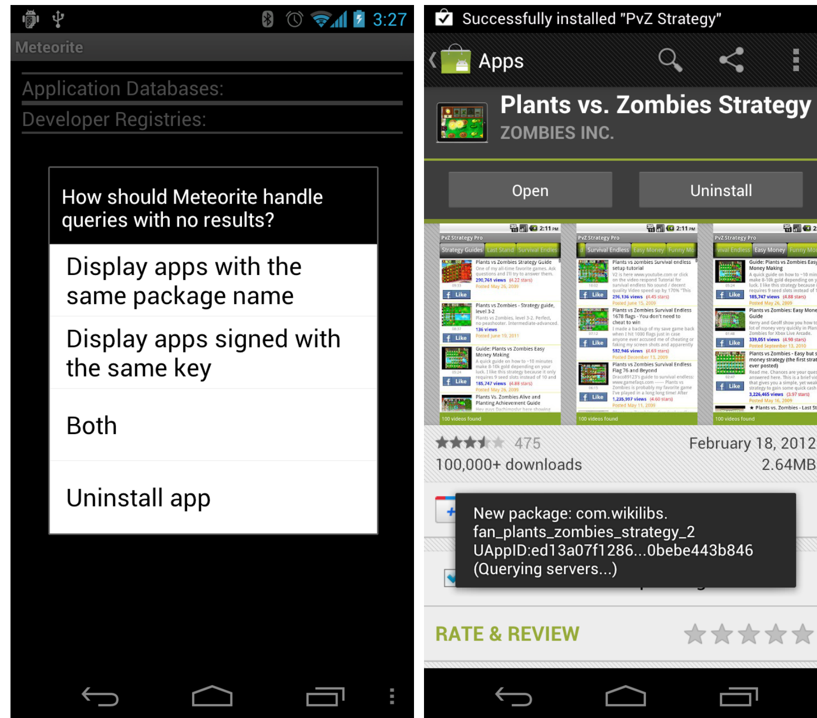
Using standard HTTP queries and a standardized query language allows information source operators to populate and run their servers in the way that is most convenient to them. Future versions of Meteor will specify an app description language that will assist in automated decision making based on a local policy. We note that our prototype Meteorite app has not been engineered for usability and is presently mainly a tool for expert users.

4.5.3 Kill Switch Listener

The final component of the Meteorite app is the kill switch listener (depicted as item 4 in Figure 4.2). We avoid polling (which is expensive in terms of battery and network performance) kill switch authorities by using Google's Cloud to Device Messaging (C2DM) Framework [60] for push notifications to devices. C2DM allows a server to send short data messages (up to 1024 bytes) to registered devices. The messages are sent from the kill switch authority to Google, and then relayed to the Meteorite app using the existing connection most Android devices maintain with Google servers.

When adding each kill switch authority to the Meteorite app, a registration process must take place to inform the kill switch authority that the user is willing to receive kill notifications. The user can remove or temporarily disable kill switch authorities from within the app, as well as choose to ignore kill switch messages when they are received.

[7] These intents are received regardless of the app's installation origin (*e.g.,* sideloaded, third party app market, official market).



(a) Configuring the Meteorite app

(b) New app installed notification

Figure 4.5: Screenshots of the Meteorite app. On the left, the settings window allows users to configure the behaviour of the app when no apps are found in an application database. On the right, a notification that a new app has been installed and a query is in progress in the background.

Our demonstration kill switch authority currently sends the UAppID of the app to be killed, along with a reason the kill switch was thrown. Receiving a kill switch launches Meteorite in the background and compares the received UAppID against a list of UAppIDs for all installed apps. If a match is found, the user's current Android activity is interrupted to display the kill switch data. The user is then prompted to either uninstall the app or dismiss the message.

As noted earlier, Meteorite does not have root privileges so it is not capable of removing other apps without the user's consent. Meteorite only invokes Android's package manager and the user must confirm or deny the final uninstall process.

4.5.4 Design Choices

The Meteorite app leverages Android's application life-cycle and IPC to avoid running in the background and constant server polling. The OS only invokes Meteorite at app installation time or upon receiving a kill switch, meaning that there is negligible impact on battery life or overall performance.

Our choice to use non-privileged APIs comes with the trade-off of creating a vulnerability window between app installation and first launch. Malicious apps could register hooks to launch automatically (*e.g.,* after a device boots, after receiving an SMS, or as a handler for specific file types). This window could be avoided if the Android app installer were modified to perform Meteor queries prior to installation, but would require installing a full new Android firmware which may discourage widespread adoption.

Source code for the Meteor app as well as UAppID utilities are available for download at `http://www.ccsl.carleton.ca/software/meteor`.

4.6 Discussion of the Meteor Architecture

Here we discuss the advantages and limitations of the Meteor architecture as a whole, and review the specific benefits of grouping information sources by domain and security focus.

4.6.1 Advantages

Incrementally deployable and extensible: The proposed architecture provides security benefits even if using as few as one information source or kill switch authority. The client-side software could initially be distributed as an app download, and later possibly bundled as a core OS feature. The ability to customize information sources offers an extensible solution that can be adapted to a wide range of users/skill levels, and facilitates the existence of a democratic application marketplace.

Scalability: By allowing experts in small domains to handle application reviews and database maintenance, high quality information may be produced and distributed quickly.

Low resource requirement: Servers are only queried upon app installation, where a small amount of application-specific data is sent and received. Meteor involves no massive (*e.g.,* antivirus) signature downloads and no CPU-heavy operations performed on the device aside from hashing the app in question.

Only intrusive on suspicious behaviour: The Meteorite app can be configured to only display warning messages in the event of detecting suspicious activity (by the developer or app). If no malicious activity has been reported for the app being installed, the installation process is not visibly different to the end-user than the current app installation procedures.

4.6.2 Limitations

We note that Meteor mitigates many threats introduced by a multiple-market ecosystem, but it cannot address all threats. In particular, our architecture cannot automatically determine the most (or least) secure application for a specific task. This is a fundamental problem of app distribution even for single-market ecosystems, and is outside the scope of this thesis.

Usability: Meteorite requires some level of user input either for adding servers or interpreting the results of the information source query. While we can attempt to automate some of these tasks (*e.g.,* by specifying policies), our proposal is not a one-size-fits-all solution. Future work is needed to identify usability challenges when deploying a system like this.

Quality of information sources: There is no mechanism to automatically handle malicious information sources, which may allow attackers to craft a complex attack where a user is tricked into installing a malicious information source as well as a malicious app. In this example, the information source could return positive comments and ratings for all apps, or at least those an attacker wishes to be installed, giving the

user a false sense of trust that the app being installed is benign (false negatives). The information source could also attempt a denial of service by replying with warnings on all queries, creating false positives which must be identified and resolved by the user. While attacks like these are less likely as more information sources are selected (assuming at least one has identified the malicious app), they may still be possible for users with small information server lists.

We believe that over time, individuals and sub-communities will place their trust in information sources that deserve such trust. While Meteor and other (*e.g.,* single-market ecosystems with state-of-the-art vetting) systems fundamentally cannot preclude abuse of trust, Meteor can deal with abuse in the longer run by user choice.

4.6.3 Security and Privacy Considerations

Separating content creators from content hosts: Meteor does not require that application databases be hosted by the same individuals who create the content within them. For various reasons (*e.g.,* high cost or unavailable expertise), expert users may not want to operate a dedicated server even though they have access to app or developer information. Similarly, end-users may trust an entity with the app information they create, but may not be willing to reveal to that entity the apps they are installing. By allowing this separation, database hosts can act as relays or proxies to help preserve end-user privacy. Meteor could be extended to allow private information retrieval (PIR) [32] for users with high privacy requirements.

Authenticating information: In its most basic form, Meteorite clients communicate with Meteor servers via HTTPS using Android's default root CA list. Future work is needed to design a proper way to handle man in the middle attacks on the local network (*e.g.,* an attacker intercepting both the app file download and the Meteor query). We are exploring the option of using signed database records that can be authenticated locally, similar to the approach of Samuel *et al.* [106].

Lack of consensus in database entries: In a distributed system like Meteor, it is possible that users subscribing to many information sources will receive conflicting information. For example, an app may exist on a whitelist for one reason and simul-

taneously be listed on a blacklist for a completely different reason. The Meteorite app currently requires consensus among information sources that are queried, but future work will look at ways to automatically resolve conflicts based on pre-defined policies.

4.7 Security Analysis

This section presents a security analysis of the Meteor architecture. Our analysis identifies threats created by allowing the presence and active participation of corrupted users, developers, information sources, and collusion between them.

Corrupted Developer: A developer may attempt to produce a malicious application which, when looked up on Meteor servers, returns information associated with a distinct legitimate application. To be successful (and assuming the developer doesn't have the original developer's signing key), the developer must create a new malicious binary which produces the same binary hash as the legitimate application. The use of a suitable hash function, as discussed in Section 4.4.1, should preclude this.

Corrupted Information Source: Meteor servers can violate their user's privacy by harvesting queries and IP addresses, thereby learning exactly what apps users have installed. To mitigate this threat users can obfuscate the source IP address of their device by sending queries through a proxy or Tor [40]. The downside of proxying queries is that servers can no longer have an accurate view of their userbase or maintain accurate usage statistics. IP address obfuscation may also make it difficult to detect corrupted users sending fake queries.

Another method for users to preserve privacy is to obfuscate submitted queries by using private information retrieval (PIR [32]) techniques. While PIR currently has high bandwidth and processing overhead, it may be a more viable solution on newer devices. Additionally, a combination of Tor and PIR could be used in privacy-critical environments [89].

Collusion between developers and information sources: We consider the case of corrupted developers colluding with corrupted servers for completeness, but note that collusion specifically should not introduce any additional threats. Corrupted

information sources can readily reply with specially crafted responses without the assistance of developers. Furthermore, servers cannot assist developers in finding a collision on binaries.

Corrupted User: Meteor was designed to protect users. In our proposed architecture, users have no direct input into the system by using the application, so the threat of a corrupted user (aside from a user attempting a denial of service on information sources) is negligible.

Lack of Consensus: When users subscribe to many information sources, they may receive conflicting information (*e.g.,* corrupted servers trying to convince a user that a malicious app is benign, or an app exists on both a whitelist and a blacklist). Meteor clients currently require a consensus among the majority of information sources they query. Future work will look at automatic conflict resolution based on policy.

4.8 Related Work

Meteor builds upon known proposals for unique file identification, cross-checking information, cloud-based malware detection and information crowdsourcing. This section reviews some of these proposals.

Kim and Spafford propose Tripwire [80], a tool for recording hashes of important system files and later detecting modifications or intrusions. UAppIDs are similar in that they involve cryptographic hashes of the app's binary code, but these hashes are not stored to detect modification (code signing already does this). UAppIDs uniquely identify and index (*e.g.,* for app lookup) app instances.

Oberheide *et al.* [95, 96] highlight advantages of offloading malware detection to the cloud such as low resource requirements and better detection coverage. The authors run multiple antivirus engines on each submitted binary, and hashes of binaries are stored to improve performance (*i.e.,* avoid scanning the same file if the result is already known). This is conceptually different from Meteor, which aggregates information from multiple expert sources on the device rather than aggregating multiple services on a central server.

A number of researchers have analyzed large collections of apps across multiple markets. The results of these experiments, including detection of malware [132], privacy leaks [44], and poor developer practices [51] would be good candidates for inclusion in a Meteor application database.

Perspectives [123] (and related projects such as Convergence [90]) uses a set of "notary hosts" which monitor web servers' public keys from multiple vantage points on the Internet. Clients query the notaries to detect man-in-the-middle-attacks or changes to public keys. Information sources in Meteor play a role somewhat similar to Perspectives' notaries, but app information is collected by more than passive monitoring (*e.g.,* experts actively trying apps and submitting reviews).

Meteor shares similarities with browser-based ad blocking tools such as Adblock Plus [2], which allow users to subscribe to ad blocking lists maintained by experts around the web. Similar to Meteor information sources, each ad blocking list filters specific types of advertisements (*e.g.,* region-specific, content-specific), allowing users to build a custom filter set tailored to their needs.

Aggregate and personalized ratings from users in a social circle can be helpful to find inappropriate apps, as users in the same social circle tend to have similar definitions of appropriateness [29]. However, detecting malicious applications generally requires experts, who may not be present in all social circles. Meteor attempts to create domain-specific services that can individually crowdsource information. However, Meteor does not specify how to deal with the problem of expert recruiting or "fame management" [41]. We defer this aspect, as well as the optional use of reputation systems [76] to each information source.

4.9 Summary

This chapter has discussed the main challenges in providing pre-installation security in platforms that offer decentralized software installation, using Android as an example. We proposed the creation and use of security information sources, developer registries, and kill switch authorities to re-gain the security semantics lost by allowing decentralized software installation. The design and implementation Meteor was described as a general software installation architecture that leverages domain-specific

crowd-sourced information from experts. We believe such a decentralized architecture is valuable not only for Android and smartphone OSs, but also tablets, netbooks, and computing devices in general as software distribution and installation inevitably converge in the direction of application markets.

Chapter 5

The Android Observatory

5.1 Introduction

The Android Observatory is a proof-of-concept information source for Meteor (see Section 4.4). This information source was originally designed to reply to install-time queries by clients to help users decide whether an application was authorized (*i.e.,* digitally signed) by the correct developer. This functionality was initially done by maintaining an association between known package names and signing certificates. If a client submitted a {signing certificate, package name} tuple that did not match any entries for that package name or certificate in the database, a warning was sent back to the client alerting them that the application's signing information cannot be corroborated.

We have updated and expanded the Android Observatory to display very fine grained application metadata and code signing information, allowing it to be used for more than a Meteor information source. This section explains design and implementation details of the Android Observatory. We also discuss how the dataset that is currently being used was obtained and curated.

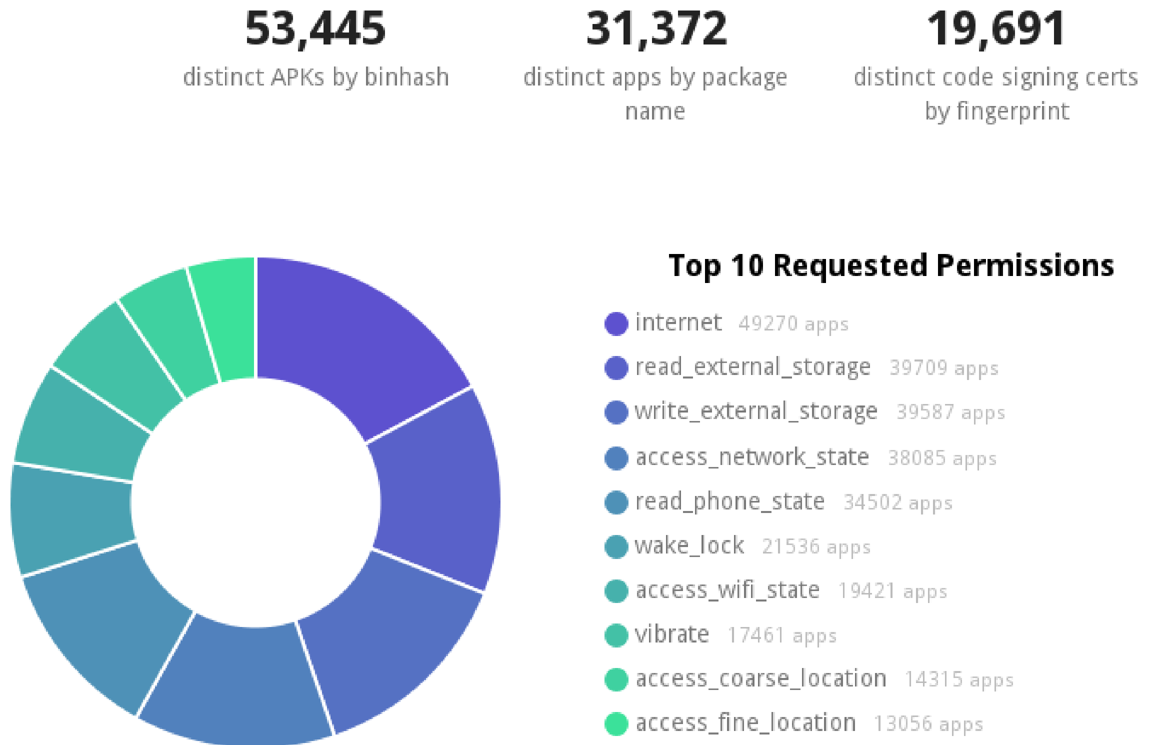


Figure 5.1: A screenshot taken from the Android Observatory website as of March 14, 2014. Statistics include number of unique samples as well as a list of the top 10 most requested permissions across all Android applications in the dataset.

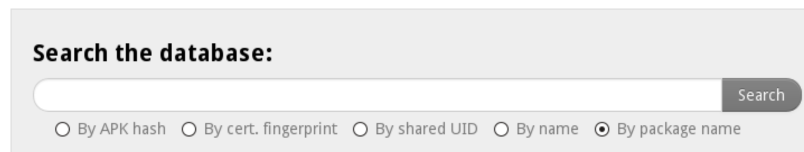


Figure 5.2: Screenshot depicting the user interface for searching applications on the Android Observatory.

5.2 Overview

The Android Observatory is a public website that displays Android application metadata. As of March 14, 2014, the Observatory contains metadata from 53,445 distinct binaries[1]. These binaries represent multiple versions of 31,372 distinct packages (see Figure 5.1). We have also indexed 19,691 code signing certificates[2]. The web front-end allows users to search (see Figure 5.2) for applications by package name, application name, and other metadata. When searching for a specific application instance (*e.g.,* by searching for a binary hash), the Observatory interface will display all metadata for that application, provided it has already been added (either directly by us or contributed by a user) to the database. Metadata displayed includes:

Application: Metadata describing each individual application's properties such as: package name, version number and version code. We also include SHA1 hashes of various application resources such as the manifest, compiled application code classes, and binary resources.

Certificates: Details about signing certificates included in the application. We include X.509 properties such as common name, organization unit, fingerprint, *etc.* We also display public key information which includes key algorithm, key size and key parameters.

Permissions: A list of all Android permissions requested by the application, including permissions that are non-existent (discussed in Section 3.4), developer-defined, and system-only.

Source: The application market or source from which the application was obtained. This information is not contained within the application itself, but rather inferred or hard-coded at import time.

On the Observatory, application metadata is displayed using hyperlinks so users can quickly navigate to other *related* applications within the dataset (*i.e.,* not on the Android market, for example). We define applications to be related if they share one or more attributes with another application in our database. For example, if

[1]Binaries are distinguished through a cryptographic hash (SHA1) of their binary code (displayed as *binhash* in Figure 5.1).

[2]Certificates are uniquely identified through a cryptographic hash of the public key included in the certificate (referred to as *certificate fingerprint* in Figure 5.1).

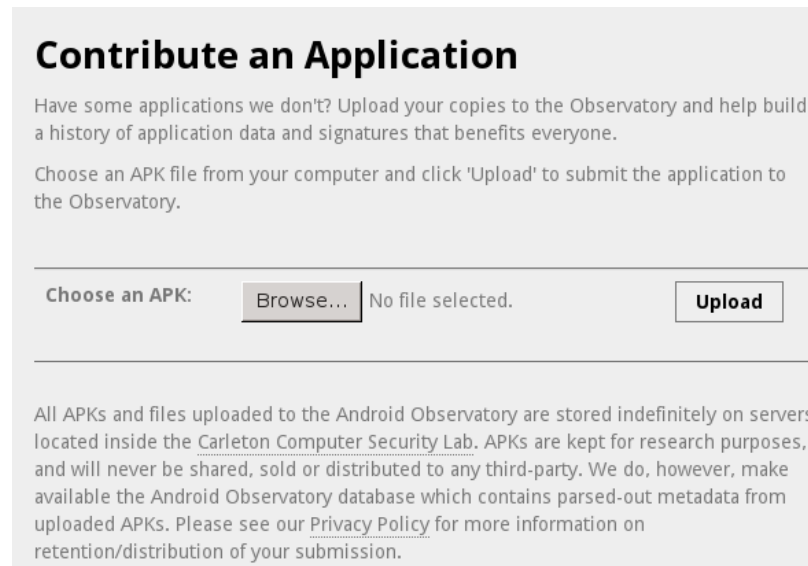


Figure 5.3: Screenshot depicting the user interface for uploading an application to the Android Observatory.

an application was signed with a signing certificate whose corresponding private key was used to sign more than one application, other applications signed with that key are also displayed. This allows users to easily navigate between apps by the same developer or development company.

Viewing apps that use the same code signing certificate has other benefits. For example, developers can detect key compromise[3]. A practical example highlighting the advantages of displaying related applications was a signing certificate[4] used on over 1500 apps in our dataset. Upon further examination, we traced back the origin of this certificate (including its corresponding private key) to being distributed as part of the Android Open Source Project as a "test key"[5] used for signing debug Android builds. Malware authors and novice developers (perhaps following incorrect guides online) have also made use of this certificate to sign applications, and the Observatory clearly highlights this behaviour.

We periodically update the Observatory database by invoking an application import script on newly obtained applications. As of writing, these applications are obtained

[3] To be more specific, the developer would have to actively visit the Android Observatory, and his or her private key would need to be used to sign a different app, and this app would also need to be submitted to the Observatory.

[4] https://androidobservatory.org/cert/61ED377E85D386A8DFEE6B864BD85B0BFAA5AF81

[5] https://github.com/android/platform_build/tree/master/target/product/security

by either crawling new and existing markets, or by user submissions via a web upload interface (see Figure 5.3). After applications are imported into the database, they immediately appear as "recent applications" on the Android Observatory website.

Advanced use

While the provided search interface is simple to use (ideally even by non-expert users), user input is converted into pre-constructed queries which are then executed on our back-end database. Thus, when using the site's search, the types of queries that can be made are limited. More complex queries can be performed directly on the database file (available for users to download) as SQLite queries; for example, it is possible to search for all apps signed with a 512 bit RSA key that request both `INTERNET` and `RECORD_AUDIO` permissions. Sharing the database of parsed metadata (but not the original application binaries) with other researchers has already lead to interesting findings beyond our original goals. For example, Truong *et al.* [116] used our database to find previously undetected malware by searching for known malicious signing keys in our database.

5.3 Application Sources

The Android Observatory contains applications collected from several sources. The project goals were to create a database that contains application representing a broad set of languages, versions, distribution models, signing schemes, benign, malicious, *etc.* Thus, we acquire applications from official and unofficial application markets, peer to peer application collections, malware repositories, and we additionally allow users to contribute their own applications or applications they have collected. Figure 5.4 displays a graphical summary of the application source distribution of our dataset. We now describe each of these application sources.

Google Play Store. The Google Play Store [66] is a factory-installed (*i.e.,* pre-installed) app on Android devices sold in regions where Google has presence. Most applications (around 6,000) in our dataset from the Play Store were obtained by

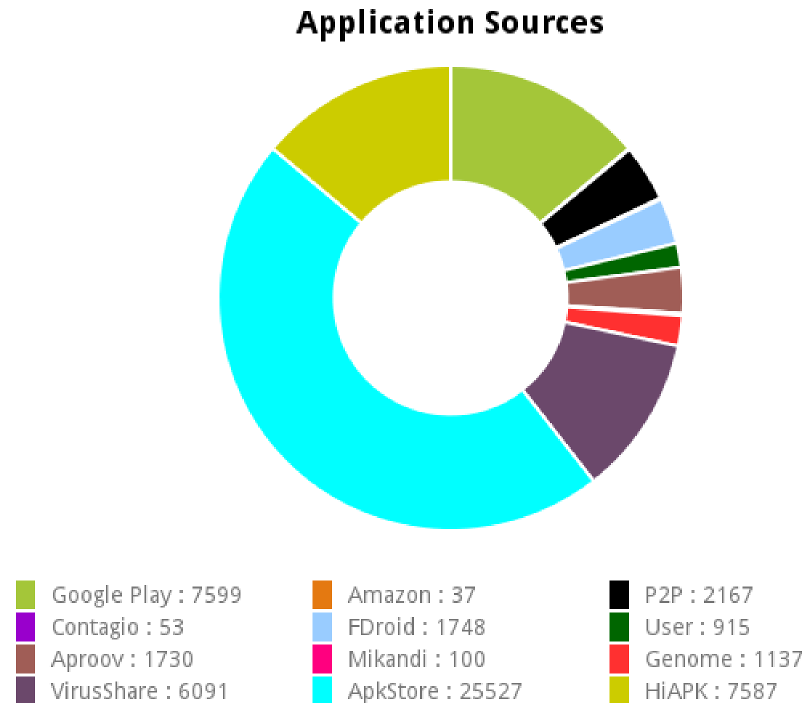


Figure 5.4: Number of Android Observatory apps from various sources.

crawling the top free applications in the Google Play Store for over a year beginning in early 2010. Another long-term crawl which contributed to our dataset took place over the summer of 2012, where we collected an additional 1,600 apps. For each of the crawls, a generic T-Mobile US identifier and region code were used as well as an Android 4.0 user agent, retrieving applications visible to US customers on those devices[6]. In total, our dataset contains 7,599 applications from the Google Play Store.

Alternative Markets. Several markets have emerged to offer alternative application delivery channels. These markets differ from the Google Play Store in their terms of service, pricing and developer payment schemes, restrictions on content, and user experience. The Android Observatory contains applications from six alternative markets.

- The Aproov market [14] is an alternative market that provides more detailed app categorization, claims greater profit shares for developers, and a more sophisticated app rating system than Google Play. The top 100 applications from

[6] Developers may select geographic regions in which to make available or disable their applications. Developers may also specify a minimum Android version required to run an application.

each of the 17 Aproov categories were included as a large non-Google Play data source. We note that as of March 2014, Aproov is no longer functioning as an application market.

- The Amazon Appstore [3] is a mainstream alternative to Google's Play Store. It is featured prominently on Amazon-branded devices such as the Kindle Fire. One paid application per day is featured by Amazon and offered for free download. We downloaded the free app of the day on 37 different days in 2012.

- The F-Droid market [48] offers only free and open source Android apps. All apps are open source (usually released under the GNU General Public Licence and Apache license) and most are built from source by the F-Droid maintainers. The authors of the applications themselves are not responsible for signing the F-Droid app package, rather they are signed by the market maintainers. We wrote a crawler to download around 1700 applications from F-Droid over 2 days in early 2013.

- The MiKandi market [88] offers applications with adult content. Adult content is prohibited on the Android market so the inclusion of these apps expands authorship in the Android Observatory's dataset. We included the top 100 free applications from July 2012. For this set and all others in the Observatory, We do not make available the original applications for download.

- The APK store market [11] appears to specialize in distributing pirated applications. Their site displays application screenshots and descriptions, and links to free file-hosting websites to download the applications. Following the download links usually leads to a copyright notice explaining that the application in question has been taken down. We were able to download over 25,000 apps from APK store in late 2012.

- The HiAPK store [74] distributes free applications in Chinese. It appears to host applications which are not available on other markets, adding diversity to our dataset. We retrieved roughly 7,500 applications from HiAPK in late 2013.

Malware Repositories. To achieve diversity in our dataset, we include meta data on malicious applications from three different sources. Each application from the sources below is labelled as malicious in our dataset (we trust that the sources of

these apps have correctly classified the app as malware, and as such are making it available to others). This way, users can identify specific malicious apps or malware related to other applications.

- Contagio Malware Dump. The Contagio website [35] hosts known desktop and mobile malware samples for research purposes. Some of the samples are contributed by anti malware-vendors, while others are user-submitted. Malware packs are periodically made available for download by Contagio, and contain a large set of malicious applications. We downloaded one malware pack from Contagio, collecting 53 separate malware applications. We note that many samples on Contagio have been independently analyzed in the literature [52, 133].

- Malware Genome. Zhou *et al.* [132] analyzed 1137 malware applications for Android, characterizing the evolution of malware for the platform. This is their dataset which was subsequently made public. We include all apps from the Malware Genome Project in the Observatory.

- VirusShare [118] is a website dedicated to collecting and re-distributing (for research purposes) malware samples for multiple platforms. In early 2013, VirusShare released a collection of 6091 malware applications for Android. We include these in our dataset.

File-sharing. In early 2012, we downloaded 4 popular app collections that were available on the BitTorrent network. File-sharing networks are a common source of pirated content and are commonly considered to present a risk of malware. The 4 collections totalled 2,167 unique binaries.

User Submissions. As mentioned, the Observatory allows any independent party to submit applications for analysis. As of March 14, 2014, we have received 915 unique user submissions. While we receive multiple submissions a day on average, some submissions are duplicates entries of applications already contained in our database. For efficiency, we discard duplicates upon submission.

5.4 Populating the Observatory Dataset

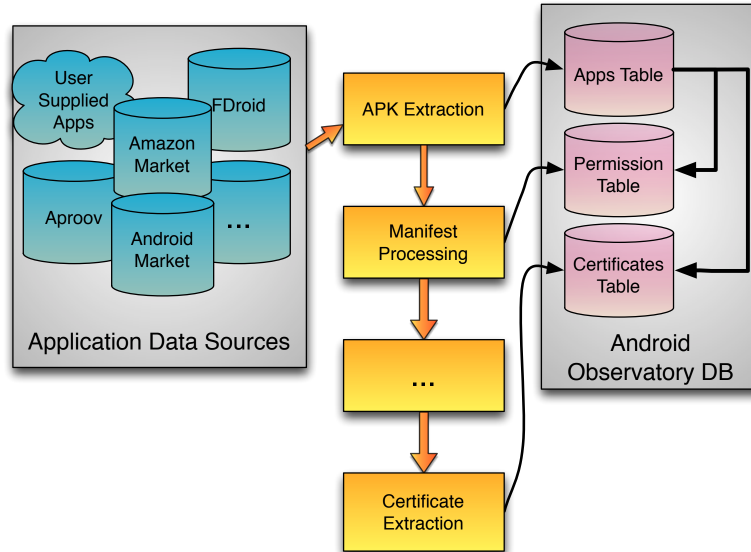


Figure 5.5: Block diagram of application import process.

The database is populated with information extracted from Android packages. We use what is formally called a chain of responsibility pipeline [56]. This type of process allows the import and processing of applications from a variety of sources. Each component of the chain (*e.g.,* certificate parser, metadata parser, cryptographic hash calculator) is able to independently process application information, insert database records, and prepare the application for subsequent components. The modular nature of the pipeline allows for new analysis operations (*e.g.,* static analysis tools) to be integrated in order to augment the database with additional information.

The pipeline extracts and computes two main types of data from each Android app package (see Figure 5.5): (1) metadata extracted from the package such as app names, package names, version numbers, permissions, and certificate attributes from file contents; and (2) a structural description of the package such as the individual hashes of resources, compiled code, and the application manifest. Distinct variants of the same app (*e.g.,* different versions or same version from difference sources) are indexed separately as unique APKs. Relevant pieces of the extracted information are stored in separate database tables for application packages, certificates, and permissions. By cross-referencing common fields in the tables, it is possible to locate common elements between applications.

In order to obtain attributes from certificates for inclusion in the dataset, we modified Oracle's Java `keytool` utility to extract more detailed information than the base tool provided. We extracted the certificate serial number, public key and public parameters (modulus and exponent for DSA and RSA).

5.5 Future Work

Moving forward, we see the Android Observatory continuing to be useful to researchers and the general Android user community. We believe it would be useful to grow the dataset used herein through the inclusion of new and emerging marketplaces, as well as increasing coverage of our existing sources. We may also improve integration of the Observatory with other online application repositories, allowing users to find applications outside of our own dataset. On each application page, we currently provide links to VirusTotal [119] and Andrototal [9] (both are cloud-based malware scanners) so users can obtain malware scan information (provided the application was independently submitted to those services for analysis). While we do not yet offer a complete API to query the Observatory, we have already shared the Observatory database of extracted metadata with many researchers.

Chapter 6

Securing Software Downloads and Updates

6.1 Introduction

After users select an application to install, possibly leveraging a pre-installation architecture like the one described in the previous chapter, the next step in the software installation process is to retrieve the application and verify its authenticity and integrity. By authenticity, we refer ensuring that an application was authored by the intended developer. By integrity, we refer to the verification that the application was downloaded (*e.g.,* in the case of remote retrieval) without error. Digital signatures play a key role in providing authentication and integrity guarantees in software distribution and installation.

Modern operating systems use digital signatures as a mechanism to verify the integrity of downloaded software and/or authenticate developers. Platforms such as iOS, Windows Phone, and Blackberry use code signatures to restrict installation of third-party applications to only registered developers. These platforms use a centralized authority, where developer certificates or the software itself is signed by the vendor prior to being distributed to user devices. A centralized authority is restrictive for users (*e.g.,* users *must* obtain software and updates from sources formally sanctioned by the platform vendor) while allowing vendor control over certificate issuance.

Android, one of the most widely deployed mobile operating systems, does not use a centralized authority. Instead, developers are responsible for obtaining suitable signing certificates (typically self-signed, but they can be issued by a certificate authority). The use of signatures on Android allows the OS to verify (1) that the application package was downloaded without error, and (2) to authenticate updates. Specifically, Android's installation framework allows installation of app updates only if they are sanctioned by the same developer. Such *update integrity* is enforced in the OS by comparing the set of signing certificates embedded in the already installed application against the set in the updated version. If the updated version set of certificates matches the set in the previously installed app, the update is allowed. Otherwise, the update fails. Since Android uses a trust-on-first-use [123] approach, initial app installations are not subject to such *certificate continuity verification*. However, download integrity can still be provided, as any missing or erroneous bits would result in digital signature that cannot be verified.

In this chapter, we motivate, design, and implement Baton, protocol that allows developers in decentralized environments to update code signing certificates (informally called *key agility* or *certificate agility*). Baton offers a mechanism to delegate signing authority to a new certificate, and provides a cryptographically verifiable mechanism to ensure the authenticity of certificate delegations. Developers may wish to update the certificate for a variety of reasons. For example, developers may want to authorize a new developer to issue updates to an application in the case of an application sale or transfer. In other cases, developers may want to use new signing algorithms or key sizes as acceptable best-practices evolve. We implement and test Baton in a real-world decentralized environment (Android), and find that certificate agility can be provided without end-user involvement, user data loss, or changes to the decentralized code signing model.

6.2 Background and Related Work

6.2.1 Download Integrity

We briefly discuss issues surrounding verifying download integrity when digital signatures are not available, and then concentrate the discussion on environments in which all software is digitally signed.

Download integrity refers to verifying that a software installation package or binary was not modified during the data transfer (typically a web download) process. That is, download integrity focuses on data *consistency*, ensuring that the file which was made available by the developer is retrieved without modification (*e.g.,* due to lack of error checking in the file transfer protocol, or malicious substitution of data by an adversary). When there is no authentication during the download process (*e.g.,* a file transfer over plaintext HTTP), integrity is typically provided by the underlying transport protocol. A common practice in signature-less decentralized software distribution is for developers to provide the output of cryptographic hash functions (*e.g.,* MD5 or SHA1) using the software package as input. The integrity of a downloaded file can be verified by retrieving a cryptographic hash of a file, and comparing to a locally computed hash as shown in Figure 6.1.

```
$ wget http://ftp.mozilla.org/pub/firefox/releases/SHA1SUMS
$ wget http://ftp.mozilla.org/pub/firefox/releases/firefox-28.0b9.tar.bz2
$ sha1sum -c SHA1SUMS
firefox-28.0b9.tar.bz2: OK
$ wget http://ftp.mozilla.org.evil.com/firefox-28.0b9.tar.bz2
$ sha1sum -c SHA1SUMS
firefox-28.0b9.tar.bz2: FAILED
sha1sum: WARNING: 1 computed checksum did NOT match
```

Figure 6.1: Example commands used to verify the integrity of a downloaded file. The `sha1sum` program takes as input a list of SHA1 hashes, calculates a new SHA1 hash on the file, and compares the results. In this example, the list of hashes is retrieved from the legitimate server, and the second binary is retrieved from a malicious site.

We note that in the example of Figure 6.1, the SHA1 hash for the file was retrieved over HTTP. An adversary could have performed a man-in-the-middle attack at re-

placed both the SHA1 hash as well as the downloaded file, allowing for successful integrity verification, but incorrect (and unverified) authentication. Thus, an additional layer of authentication (*e.g.,* one provided by a digital signature) is needed to ensure the cryptographic hash can be trusted. Alternatively, the SHA1 hash could be retrieved out of band (*e.g.,* posted on a mailing list archived on a different server), but that process is beyond our current scope.

6.2.2 Digital Signatures for Authenticating and Verifying Downloads

Digital signatures are the most common solution to verifying download integrity and authenticity when a public key infrastructure is available. In a digital signature scheme, a cryptographic hash of the message (in this case, the file being made available for download) is computed. The hash of the message is then processed according to a digital signature scheme[1] and a private key to produce a digital signature. Upon reception of the file and corresponding signature, the receiver computes the hash of the file, and verifies the signature using a signature verification algorithm and the sender's public key. If the signature correctly verifies the receiver can at least make the following two claims:

1. Authenticity: the signature was produced by someone in possession of the sender's private key. Since private keys should remain private, the receiver can be assured that it was in fact someone with possession of the private key who created the signature.

2. Integrity: The signature was created with the inclusion of a cryptographic hash of the application. Thus, if the file was modified in transit, the signature would not verify correctly. If a malicious party attempts to modify the application in transit, they would also need to generate a valid signature. This should be difficult without possession of a private key.

[1] Many digital signature schemes exist, each with its own performance and security properties. In this thesis we treat the signature scheme as a black box that could internally use any digital signature algorithm that requires the message as input for verification. Readers interested in the inner workings of various digital signature schemes are encouraged to read Chapter 11 of the Handbook of Applied Cryptography [87].

We note that confidentiality is one property that is not inherently provided by the use of digital signatures. That is, an eavesdropper that can monitor the channel used to send the application will be able to identify the sender, as well as completely view the application's contents. Confidentiality can be provided by encrypting the file itself or using an end-to-end secure channel.

6.2.3 Trust on First Use in Decentralized Environments

Trust on first use (TOFU) is used when there is no central authority in charge of certificate issuance or revocation. In a TOFU model, certificates are trusted the first time they are seen, hoping that the certificate was not forged, tampered with, or substituted. Thus, TOFU schemes are also referred to as leap-of-faith authentication [16]. Note that in a TOFU scheme, certificates does not fully authenticate their owner since no authoritative party verifies the link between identity and public key. Goldberg *et al.* [58] categorize TOFU protocols as stateless, non-interactive message recognition models that can be used to authenticate messages as long as the same certificate is used to sign every message. TOFU protocols are lightweight, and require no certificate authorities or centralized PKI of any form (*cf.* [6, 85, 58]). Their drawback, consequently, is the inability to provide trust guarantees on the initial verification, as well as locking-in trust to a specific certificate after the initial verification.

Android, as described in Section 3.5, requires digital signatures on all applications. Android uses the TOFU approach, providing no trust guarantees on initial installations, but allowing only applications signed with the same certificate to replace existing applications. When signing certificates during updates are changed, the only method to install the updated app is for the user to manually uninstall the old app (which deletes the app's user data) and then install the updated version as a new installation which is not subject to the certificate continuity verification. This process in effect revokes trust in the previous signing certificate, replacing it with one that is newly trusted for subsequent updates.

Aside from this uninstall-reinstall method described above, Android, in its operating system and developer tools, has no mechanism for developers to renew, change, or revoke signing certificate(s).

6.2.4 Selective Updates

Wurster *et al.* [127] propose a mechanism for providing binary file update integrity through self-signing by including an embedded set of public signing keys, sufficient for verifying signatures. This approach is functionally similar to Android's app update mechanism, and similarly does not allow updating or changing verification keys. In subsequent work [117] (called *key-locking*), the verification key set is allowed to evolve, meaning that updates may include new verification keys suitable for verifying the next set of updates.

The key-locking approach presents several advantages over the TOFU model. First, it allows signers to update their signing keys in a flexible way. For example, a threshold scheme can be done by issuing an update that requires t-of-n signing keys to be present in the subsequent updates. The total number keys can grow or shrink as needed, as long as updates are applied in order to preserve continuity. Second, key-locking is transparent to users. The OS enforces verification of signatures, and installs new keys for future updates. If signatures are valid, updates are applied without warning or input from the user.

The disadvantage of key-locking is that all intermediate updates must be applied for continuity. That is, if an intermediate update N in which keys are changed is not applied, the OS may prevent the installation of the $N + 1$ update due to verification keys for the update not being present in the $N - 1$ binary. This limitation can be side stepped by including verification keys over multiple updates, but this may be undesirable in the event of key compromise.

6.2.5 Other Related Work

In the broader literature on software updates, Cappos *et al.* [28, 27] examine security issues in package managers which commonly distribute verifying keys as part of the installation media. Samuel *et al.* [106] describe a software update framework (TUF) that is resilient to a number of key compromise attacks. TUF is essentially an alternate PKI tailored to allow multiple roles (*e.g.,* release and time-stamping) such that an adversary would have to compromise multiple keys to trick a user into installing a malicious update. Our proposal extends the TOFU model and focuses

on the continuation and delegation of the initial trust without the need for a central PKI.

6.3 Motivation for Key Agility on Android

This section presents arguments in favour of enabling key agility on Android backed by an empirical dataset obtained from the Android Observatory project (see Section 5).

6.3.1 Absence of Secure Defaults

The Android developer tools provide a point-and-click wizard for signing applications. The wizard requests (as input from the developer) a certificate validity period (Google requires a validity of 25 years or more for apps submitted to the Play Store [61]), and then invokes Java's `keytool` to generate a suitable signing key and certificate. Parameters such as key type, key size or signing algorithm cannot be specified using this wizard. However, it does pass a `-keyalg RSA` parameter to `keytool`, generating a default 1024 or 2048 bit RSA key (on Java 6 and 7, respectively). When invoked outside the wizard (*e.g.,* from the command line) without any parameters, `keytool` generates a 1024 bit (in a 160 bit subgroup) DSA public key.

On a dataset of 6159 signing certificates obtained from a snapshot of the Android Observatory from September 6, 2013, we observe that over 99% of certificates were likely generated using the Android signing wizard (see Table 6.1). In fact, only 24 certificates appear to have been generated by passing manual (non-default) options to `keytool`. According to recommendations by the National Institute of Standards and Technology (NIST), signature generation with key sizes less than 2048 bits for RSA and DSA was deprecated in 2011 and disallowed in 2013 [19, 18]. By this recommendation, over 75% of keys in our dataset do not follow best-practices, yet developers have no mechanism to transparently issue an updated certificate with a stronger key. Enabling certificate agility allows developers to change key algorithms or key sizes as best practices evolve.

Key algorithm/size	Occurrences	% of total
RSA 1024	4593	74.57
RSA 2048	1340	21.76
DSA 1024	202	3.28
Other (non-default)	24	0.39

Table 6.1: Signing key algorithm and key size over a dataset of 6159 certificates from the Android Observatory.

6.3.2 Ownership Transfer

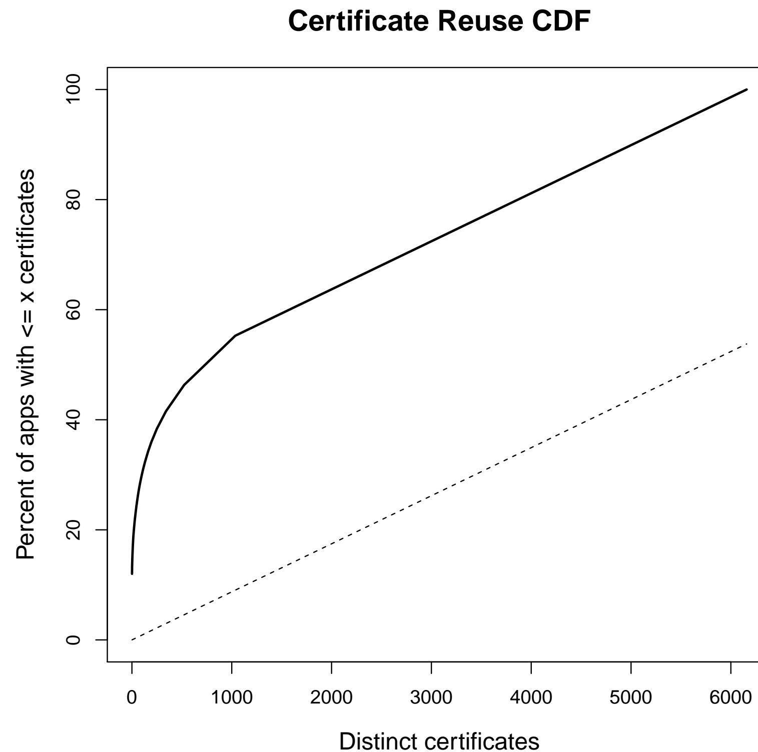


Figure 6.2: CDF for certificate reuse in our dataset of 11452 apps and 6159 certificates. An $x = y$ line depicting a theoretical 1 distinct certificate per distinct app is plotted for reference.

Applications can be sold or otherwise transferred between developers. Under the current model and to avoid user interaction, *transferring private signing keys* is a likely component of the ownership transfer process. However, such sharing of private signing keys is problematic if a developer signs multiple apps with the same signing certificate; surrendering a private key for one app allows the new owner of the key to

issue updates to any other app signed with that key.

The September 6, 2013 snapshot of the Android Observatory consists of 11452 distinct[2] applications. On this dataset, we notice that key reuse (*i.e.,* using one key to sign more than one distinct app) happens frequently. Out of a total 6159 signing certificates, 1037 keys (16.83%) were used to sign 2 or more distinct apps. Figure 6.2 shows a cumulative distribution function of signing key reuse on our dataset. Some cases of reuse involve developers releasing a free (usually ad-supported) version of an app alongside a paid version signed with the same key. In other cases, software companies (*e.g.,* Rovio, Google, Yahoo, *etc.*) release a separate app for each service or game, but use a single signing key on all apps.

6.3.3 Logical Requirements

Secure app interaction. Developers (or development teams) can leverage Android's signature-level privileges for UID sharing or signature permissions (see Section 3.6) to securely integrate apps or app components. Without certificate agility, developers must decide ahead of time which apps should be granted the capability to interact securely. It may not always be possible to predict future functionality or required interaction of an application.

External certificate management Developers may wish to use certificates (perhaps previously acquired) issued by a certificate authority to assert a validated identity on their apps. However, many reputable CAs will not issue certificates with an essentially infinite lifespan (25 or more years). Some CAs (*e.g.,* Symantec [111]) will issue these long-lived certificates but will not release them to the developer; the developer must use a CA-provided signing service to sign applications, which is an additional cost and necessitates distinct certificates specifically for Android apps. Without certificate agility, developers cannot renew an expired certificate and still update user apps without user interaction.

[2] We use the package name to differentiate between apps and to avoid counting app updates as key reuse.

6.3.4 Case Studies

This section describes two high-profile examples highlighting the potential benefits of certificate agility on Android.

Google Authenticator In March 2012, Google changed the signing certificate for their two-step authentication app Google Authenticator (package name `authenticator`[3]). Google released a new application (under package name `authenticator2`) signed with a new signing key and included a certificate used to sign other prominent Google properties (*e.g.,* Maps, Chrome, and the Play Store client). The certificate switch was ostensibly required to enable secure interaction (see Section 3.6) between Authenticator and this set of apps.

The upgrade path from one version of Authenticator to the other required that users take a series of steps, including a manual new install and uninstall. To assist users, Google created a help page [62] explaining the upgrade procedure. Below is an excerpt:

> *"Once you have confirmed as part of the previous step that you are able to successfully generate valid verification codes using the new Authenticator, it is safe to uninstall the old version of the app. Because both versions have the same icon, make sure to check the version number before uninstalling: you want to keep version 2.15."*

In Appendix A, we perform a usability analysis technique known as a cognitive walkthrough [124] on this upgrade process. We find that the overall process is convoluted and should not involve the user. However, given the constraints, Google did mitigate many potential usability issues. With Baton, we aim to provide a mechanism by which, when developers update apps which include changed signing certificates, no additional interactions are triggered for end-users when the updated apps install. Baton would have allowed Google to issue a standard update to `authenticator` which includes the new signing certificate.

Mozilla Firefox for Android Before releasing Firefox for Android in 2010, Mozilla's intention appeared to be to use their existing Microsoft Authenticode certificates or

[3] The full package name is `com.google.android.apps.authenticator2`

to purchase a 2 or 3 year certificate from Verisign to sign Firefox for Android [91]. Mozilla correctly concluded that there is no support in Android for certificate renewal, even if there is no change to the signing key pair. Mozilla filed a bug report [10] on Android asking for confirmation or motivation for why certificate renewal is not supported. The bug report was closed automatically getting marked as obsolete on June 23, 2013 despite the issue remaining unresolved and unacknowledged.

While the Android OS does not currently enforce certificate expiration (*i.e.,* apps with expired certificates can be installed as usual), the Android documentation [61] asserts that certificate validity is verified. This inconsistency leads us to believe that certificate expiration policies on Android may change in the future. Baton allows developers to use shorter lived (more typical lifespan) certificates, and update them as needed by issuing an application update. This may prove useful for companies, such as Mozilla, that already have (or wish to have) code signing certificates issued by certificate authorities.

6.4 Design and Implementation of Baton

Baton provides the ability for developers to delegate signing authority to a new private key. This is accomplished by creating a data structure (token) in which the old signing key is used to sign the new certificate and additional corresponding meta data. Each token is embedded in a certificate chain describing the history of delegations. The chain is cryptographically verifiable, and embedded inside the APK file of subsequently released Android apps after the first delegation occurs. The certificate chain and verifying code are implemented to meet the following design objectives:

1. **No user involvement.** Certificates and signatures are system-level components that need never be visible to the user. Baton provides a system-level mechanism to validate certificate changes and does not involve the user in any decisions or actions.

2. **Compatibility with Android's security model related to application signing.** Android uses certificates for software update continuity and for application interaction (see Section 3.6). Baton does not change the requirements for signature permissions and UID sharing.

Protocol 1: Baton signing key endorsement protocol

Overview: The holder of $KeyA$ wishes to delegate signing authority to a new key $KeyB$.

Variables:
$KeyA, KeyB$ - the private keys corresponding to the public keys in $CertA$ and $CertB$ respectively.
$CertA, CertB$ - the signature verification certificate used to verify signatures on current application release, and the certificate being delegated to, respectively. The certificates are self-signed.

Pre-requisites: The fingerprint of $CertB$ has been communicated to the holder of $KeyA$ over a channel with guaranteed integrity.

Protocol:
1. Holder of $KeyA$ generates *token*=Sig_{KeyA}{H(pkg name, version code, $CertA$, $CertB$ fingerprint, previous token hash†)}.
2. *token* is communicated to holder of $KeyB$.
3. Holder of $KeyB$ includes token in *AndroidManifest.xml* when releasing updates signed with $KeyB$.

†: If there is no previous token to hash (*i.e.,* it is the first token to be included in a certificate chain) null may be substituted for the previous token hash value.

3. **Minimal OS changes.** We add code to the Android application installation framework and developer tools, but make no other software modifications, and require no change of behaviour by developers if certificates don't need to be changed.

4. **Backwards compatibility.** Baton supports incremental deployment with incremental benefit. Users with Baton-enabled Android will be able to upgrade applications that have changed their signing certificates (provided verification of the delegation succeeds). Users without a Baton-enabled Android build can still install and upgrade applications that include Baton certificate chains. These users, as with current Android, will be unable to transparently apply software updates if there is a certificate change; instead they must uninstall the current version and install the update as a first install.

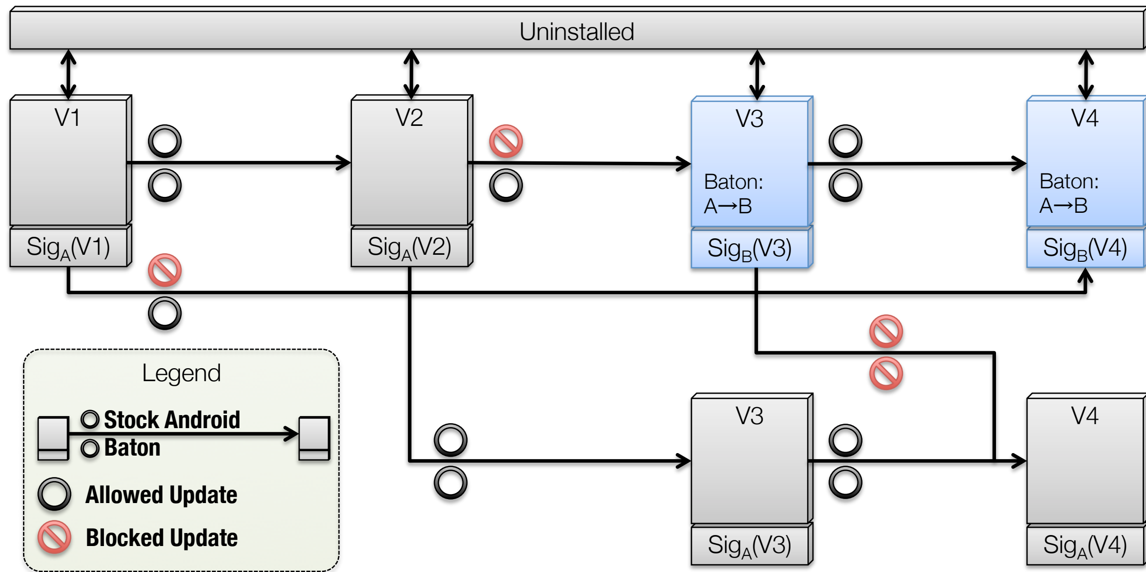


Figure 6.3: Version update diagram depicting updates that are allowed by stock Android (icons depicted over an arrow) and by Baton (icons depicted under the arrow). $\text{Sig}_A(Vn)$ and $\text{Sig}_B(Vn)$ are signatures on application version n with signing keys A and B, respectively. In all cases, updates are only allowed if signatures are successfully verified.

Baton has two core components: (1) a set of patches to the Android installation framework, modifying packages responsible for parsing the *AndroidManifest.xml* file and verifying application signatures; and (2) an Eclipse plug-in for assisting developers in generating key delegation metadata.

Delegating Signing Authority For a developer to successfully delegate (*i.e.*, endorse a new signing key) signing authority to a new signing key (shown as an example in Figure 6.3 as an update from V2_{SigA} to V3_{SigB}), they must embed a valid delegation token in the update. In the example in Figure 6.3, a delegation token passing signing authority from $KeyA$ to $KeyB$ (*i.e.*, the private keys associated with certificates A and B, respectively) must be present in the update. The delegation token generation is described as step one in the signing key endorsement protocol given in Protocol 1.

6.4.1 Threat Model and Goals

We consider the following security objectives to be necessary for any secure update mechanism, including Baton:

1. **No Unauthorized Updates.** Updates to installed apps must be authorized (either directly or transitively) by the signer(s) of the originally installed version of the app.
2. **No Replays.** Key delegation tokens should be bound to specific applications and versions. The tokens should not allow unintended delegations through embedding potentially modified tokens on unauthorized applications.
3. **Mitigating Social Engineering.** The update mechanism should only require user actions that are easily distinguishable from the actions a target victim user would take in a social engineering attack.
4. **No Unauthorized App Interaction.** Multiple apps may only interact through properly authorized privileged means (*e.g.,* sharing a UID or granting access to restricted APIs) with the mutual authorization of all the integrated apps.

We assume the attackers in the system to be computationally-bounded adversaries, who may hold their own signing keys, have their own apps released on application markets, and even have apps installed on a target user's phone. We assume adversaries are not capable of learning the private signing keys of other developers (we discuss key compromise in Section 6.6.3), nor are they able to modify or otherwise compromise the Android OS. We assume, however, that the adversary can tamper with any Android application package. A security analysis (see Section 6.5.3) is given after first describing the details of Baton.

6.4.2 Implementation

Certificate Chain and Delegation Tokens

In Baton, a *certificate chain* is a sequence of one or more delegation tokens. Each delegation token in the certificate chain is a signed collection of metadata which contains the following information:

1. The application package name.
2. The application version code.
3. A set[4] of previously active certificates.
4. A set of currently active certificates.
5. A cryptographic hash of the previous delegation token in the certificate chain.

A Baton delegation token acts as a verifiable endorsement of a transition from one set of certificates to a new set of certificates whose corresponding private keys will be used to sign the new or current version of the application. The generation of the delegation token is described in Protocol 1. Each delegation token, including a signed hash of the delegation token prior to itself in the chain, allows cryptographic verification of the entire certificate chain. This prevents an adversary from removing, adding, or rearranging delegation token elements in the chain. Inclusion of the package name scopes the delegation to only the specified application. For example, if a developer signs three applications with the same signing key and generates a Baton delegation token to update the certificate of only one of the three applications, the scope prevents this same token from being embedded in the other two applications, as the package name will not match.

Baton XML

Baton applications embed into the *AndroidManifest.xml* an XML representation of the certificate chain. To simplify the signing and verification procedure we detach [5] the delegation token signatures from the delegation token metadata to create two separate sets of nested elements, `certificate-chain` and `certificate-chain-signatures` (see Figure 6.4). The `delegation-token` elements in `certificate-chain` are matched to the corresponding `delegation-token-signature` elements in the `certificate-chain-signatures` by order within their parent element. The signing process is performed following the `xmldsig` standard best practices outlined by the W3C working group [120].

To allow signature validation in the case of missed intermediate updates, each delegation token includes a Base64 encoding of each certificate in the previous certificate set as well as their fingerprints. We chose to embed the full certificate for each of

[4] Baton assumes developers may use multiple signing keys on the same application (see Section 3.5.1).

[5] This is done by using the XML Signature Detached specification as outlined by the W3C [121].

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   >
 ...
  <certificate-chain>
    <delegation-token android:versionCode="10">
      <previous-certs>
        <cert encoded="..." fingerprint="907
            EB3F2E8447054446A2A4B3ED8CA78DB04B188" />
      </previous-certs>
      <current-certs>
        <cert fingerprint="6E532E87A468052DA2EE8E9D6E56080181D3E2F9"
            />
      </current-certs>
      <previous-token hash="null" />
    </delegation-token>
  </certificate-chain>

  <ceritificate-chain-signatures>
    <delegation-token-signatures>
      <Signature>
        ....
      </Signature>
    </delegation-token-signatures>
  </certificate-chain-signatures>
  ...
</manifest>
```

Figure 6.4: Example Baton certificate chain entry in *AndroidManifest.xml*. Base64 encoded certificate content and non-Baton related Android entries removed for brevity. Signature element defined in `xmldsig` [121] standard.

the `previous-certs` as a convenience to handle updates from a very old version to a new version signed with certificates which would be valid only after processing several delegations. In this case, the certificates specified in intermediate tokens may not be present within the application and must be loaded from the encoded version in the token.

As a design alternative, it is possible to avoid embedding a certificate chain at all by retaining all previous versions of *AndroidManifest.xml* and the `META-INF` directory in future versions of the APK.[6] Thus, signing a new APK will bind the current *AndroidManifest.xml* to the complete history of previously signed *AndroidManifest.xml* files. In this alternative implementation, transitioning to a new set of signing certificates would require the *AndroidManifest.xml* to specify the new certificate in an APK update signed by the currently valid certificate. This simpler implementation would require more involvement from developers to ensure that all prior versions of the `META-INF` directory are retained. `Jarsigner` would need to be invoked independently once the application archive is created. With Baton, certificate delegations only add a few lines to *AndroidManifest.xml* instead of creating an archive of past files containing mostly no-longer valid data.

AOSP Implementation

We modified The Android Open Source Project (AOSP) code to implement the Baton certificate verification functionality. The proposed set of patches totals under 500 lines of code which we plan to make available under an open source license compatible for inclusion in AOSP.

The `android.content.pm.PackageParser` core class was modified to correctly process of the new *AndroidManifest.xml* entries. In the AOSP services sub-project, the `com.android.server.pm. PackageSignatures` class was modified to store a *SignatureChain* reference, populated by the `PackageParser`. When the `com.android.server.pm.Settin` class loads or stores the on-disk *packages.xml* file, the *SignatureChain* is responsible for writing its own representation to the file, and restoring it when the operating system boots.

In addition to modifying existing classes, we created a new class (`com.android.server.pm.Signa-`

[6] The hash tree structure of `MANIFEST.MF` allows the signature on a single file to be verified.

`tureChain`). This class serves as an in-memory representation of the XML certificate chain from the *AndroidManifest.xml*. It contains the logic for reading the *SignatureChain* to and from XML, as well as verifying delegation token signatures.

Finally the `com.android.server.pm.PackageManagerService` class was modified to instrument the stock signature verification logic and package update procedure. When the modified `PackageManager Service` processes an update for an installed application, it will now compare the installed application's set of signing certificates to the proposed update's set of signing certificates. If the sets match, the update proceeds following the existing Android certificate continuity policy. If the certificate sets do not match, then the `PackageManagerService` ensures that the proposed update must contain a delegation token for the correct version transition (*i.e.,* for the currently installed version to the update's version) endorsed by the installed application's certificate set. The version transition may be endorsed through one or more intermediate delegation tokens allowing the update to proceed in the event the user has missed interim updates.

The AOSP project does not include the `javax.xml.crypto.dsig` packages used to verify XML signatures. Therefore we additionally include the Apache Santuario [113] library, an independent implementation of the `xmldsig` [121] standard.

Developer Tools

To facilitate adoption by developers, we augment the application signing life-cycle by integrating with the development environment used to produce Android application releases. As of writing, the official Android Developer Tools (ADT) plugin for Eclipse [59] is closed source, impeding our ability to enable Baton support directly. In lieu of a patch to ADT, we have opted to provide a third party Eclipse plugin. After installing the Baton plugin in Eclipse, developers are able to export their Android projects as a Baton-enabled APK. In addition to the Eclipse plugin, our developer tools can operate as a stand-alone GUI, or a command line tool. The stand-alone versions of the plugin are better suited for integration with other IDEs that support external tools, or with more complex build management systems often used with large software projects.

To begin the process, the developer is prompted to select one or more signing certifi-

cates to endorse for future updates, or to enter a certificate fingerprint. For example, this may be a certificate generated by the new owner of the project if the developer is transferring control (*e.g.,* selling to another developer) of their application. The certificate may also be locally generated. The developer must also choose one or more signing certificates whose corresponding private keys will be used to sign the delegation token. In practice the developer will most often select the signing certificates presently used to sign production releases of their project. The version code and the package name values in the token are pre-filled from the values in the *AndroidManifest.xml* file. If necessary, the developer will be asked to enter a passphrase to unlock the private key. After unlocking the private keys (if required) the XML for the delegation token is generated, following Protocol 1. It is signed, and inserted into the certificate chain in the *AndroidManifest.xml* file. The token is also displayed on-screen to allow communication of the token to other parties if required.

6.5 Evaluation

This Section discusses compatibility of Baton with other signature-based security mechanisms and previous versions of the OS. Additionally, we perform a security analysis of Baton.

6.5.1 Compatibility

Compatibility with Related OS Functions

Android currently uses code signing certificates for security operations outside of application update integrity. Code signing certificates provide a form of access control, selectively allowing applications to join a shared UID group or to be granted a signature permission (see Section 3.6). At application install time, for the application to be allowed access to a signature protected resource (*e.g.,* UID group or permission), the certificates on the application must be identical to those associated with the protected resource. This requirement remains unchanged in Baton.

Using Baton, if one application in a shared UID group or an application providing a signature protected permission transitions to a new set of signing certificates, the certificates associated with the group or permission within the OS are updated to reflect the transition. As a consequence, future updates to other applications in the UID group, or requesting the signature permission must also transition to the new set of signing certificates.

We have designed Baton such that developers cannot issue a certificate transition to one application that "evicts" other applications from a UID group by changing the associated certificate set. This behaviour is consistent with Android's current model, where applications cannot arbitrarily change UID groups during updates.[7] An eviction would, by definition, change the UID of the evicted application, leading to an inconsistent state. Our design instead honours the membership of already installed applications until they are updated. This prevents previously functional applications groups from losing functionality (by evicting a member) while introducing no detrimental security properties.

If UID group evictions are required, a more flexible mechanism is required. We propose one such mechanism called Fluid in Section 7.4.2.

Enabling Compatibility with Stock Android

Application releases that perform a certificate transition using Baton must package a modified *AndroidManifest.xml* containing a Baton certificate chain in the released APK. For this reason, we consider the compatibility of the modified application release with existing versions of Android (*i.e.,* those without Baton).

At application install-time, the Android OS parses the *AndroidManifest.xml* using the `android.con- tent.pm.PackageParser` class (located within the frameworks directory of AOSP). Once patched to enabled Baton support, the Android OS is aware of the new XML tags introduced for the certificate chain (see Figure 6.4), and can react accordingly. The default unmodified behaviour of the `PackageParser` class, as of the time of writing, sets an internal `RigidParser` constant to **false** causing

[7] https://android.googlesource.com/platform/frameworks/base/+/d0c5f515c05d05c9d24971695337daf9d6ce409c

unrecognized XML tags to be skipped without error. Based on the behaviour of the code in AOSP, if an application carrying a delegation token is installed on an unpatched OS, the certificate chain will be ignored. The application will still install correctly pending successful (stock) certificate continuity validation.

If the Baton patches were merged into AOSP, we recommend developers who release application containing a Baton certificate chain use the `android:minSdkVersion` parameter in *AndroidManifest.xml* to preclude install on systems lacking Baton support. It seems unavoidable that users without Baton support may only install application updates signed with a different set of signing certificates by first uninstalling the old application.

6.5.2 Implementation Evaluation

Standard Update

We verified that our patches to Android's installation framework did not interfere with standard update functionality by creating multiple releases (each with an incremental version code) of a test application, and side-loading each version on to an emulated Android environment in various orders (*e.g.,* V1→V2→V3, V1→V3, V2→V3→V1). All releases were signed with the same signing certificate and used the same package name. As per stock Android policy, updates succeeded but downgrades did not. The user experience was no different in the Baton environment and in the unmodified Android environment.

Certificate Agility

We created a sample application with an embedded Baton certificate chain and tested delegating signing authority from one certificate to another (keeping the package name the same). After installing the application signed with one certificate, the app was transitioned to a new certificate by embedding a token generated by the Baton developer tools in an update. Baton successfully validated the delegation token, and user data related to the test app was preserved and accessible by the application with the new certificate. We also tested changing certificates but not including the Baton

certificate chain, as well as changing certificates and including an invalid certificate chain. These updates failed with the "failed inconsistent certificate error" thrown by the Android OS as expected.

AOSP Unit Tests

We ran the bundled unit tests for the `PackageManagerService` class. These unit tests are included with the AOSP source code and are used for automated testing and to prevent any bugs from being introduced to previously functional code. We checked that the Baton system introduces no such regression errors by running the unit tests and verifying that a Baton patched system to pass the tests without failure or warning.

AOSP Code Inspection

We searched the AOSP source code tree looking for references to signing certificates. Code found interacting with the `PackageManagerService`, or with the `Signature` objects used internally to represent code signing certificates was manually inspected and examined for conflicts with Baton. No conflicts were discovered.

6.5.3 Security Analysis

Here, we analyze Baton under the security objectives and threat model presented in Section 6.4.1.

1. **No Unauthorized Updates.** Baton does not modify the requirements of Android's standard certificate continuity verification. Baton only introduces cryptographic verification of certificate chains. Thus, with Baton, an adversary must still compromise a developer's private signing key to issue an update or create a valid certificate chain to transition to a new signing certificate.

   The certificate chain and delegation tokens in Baton are included in the *AndroidManifest.xml*. They are not secret; digital signatures provide integrity

protection. Deleting the chain or delegation tokens inside the *AndroidManifest.xml* has the same effect as removing a signature from an Android package (deleting the `META-INF` directory, also known as signature stripping [20]). Apps without a Baton certificate chain or `META-INF` directory will fail to validate as legitimate updates and will not succeed in replacing an installed binary.

2. **No Replays.** Delegation tokens include a package name, version code, and are digitally signed. Replaying a delegation token on a different application (*i.e.,* copying the relevant section of the *AndroidManifest.xml* file into an application with a different package name) causes Baton to fail the certificate chain verification. Certificate transitions do not succeed unless all tokens in the chain reference the package name being updated, and corresponding signatures can be verified.

3. **Mitigating Social Engineering.** With Baton, users apply app updates as usual. However, unlike with stock Android, there is no legitimate reason to require the user to manually uninstall applications for the purpose of a key update. Training users that sometimes this action may be required (which is the case, as of writing) can lead to social engineering attacks by malicious developers; Baton eliminates the need to do so, reducing this risk.

4. **No Unauthorized Interaction.** Baton does not modify UID sharing nor signature permission requirements. Applications must be signed with the same signing key(s) at install time to leverage signature privileges (see Section 6.5.1). It is not possible to leverage Baton to arbitrarily join a UID group for which a key is not held.

6.6 Discussion

6.6.1 Limitations of Existing Proposals

The concept of forward certificate chains, where an old key signs a new key (or alternatively, the new key is transmitted over a channel secured with the old key) is a long-known key-management technique, almost as old as public key certificates,

	Enables Key Agility	Allows Skipping Updates	Compatible with Jarsigner	Incrementally Deployable	Allows Revocation
Stock Android [61]		●	●	●	
Baton (Sec 6.4)	●	●	●	●	○
Self-Signed Executables [127]		●		●	
Key-locking [117]	●			●	
Baton Alternative (Sec 6.4.2)	●	●		●	○
Central Certificate Authority	●	●	●		●

Table 6.2: A comparison of proposed update integrity mechanisms, including ones with key agility.

with applications to encrypted email [134], TLS certificates [47], and Linux files [117]. However, despite their perceived theoretical contributions, forward certificate chains have not seen widespread use. In Table 6.2, we compare decentralized signature schemes to Baton. We add to the comparison using a full fledged centralized PKI.

Baton appears to be the only proposal that offers key agility, allows skipping intermediate updates, is directly compatible with `jarsigner` and partially enables certificate revocation (see Section 6.6.4). We note that using a centralized certificate authority affords many of the same benefits provided by Baton, but would require substantial changes to the Android OS, developer tools, and overall signing model.

6.6.2 Certificate Expiration

The Android OS currently ignores the validity of signing certificates at install time, despite official documentation stating otherwise [61]. As of Android 4.2, we have verified that it is possible to install (without warning or user intervention) apps with a signing certificate that has expired. Additionally, the Google Play Store requires that all apps submitted carry a certificate valid for at least 25 years, making expiration verification redundant for marketplace installations. With Baton, certificate renewal

becomes possible, which re-enables the possibility of enforcing certificate validity. Enforcing expiration may limit the impact of key compromises (see below) and allow the optional use of CA-issued signing certificates that have more generally acceptable validity periods (*e.g.,* 1–5 years).

Baton could be modified to limit the time during which an expired certificate can be used to authorize a key delegation. For example, limiting the ability for an expired certificate to authorize a key delegation one year after expiration. This mechanism can help reduce the exposure window where an adversary can gain access to an expired private key and roll it over to a new malicious key, while giving developers ample opportunity (*e.g.,* one year) to acquire and authorize new certificates after expiration.

6.6.3 Private Key Compromise

In the current Android security model, if a developer's private signing key is compromised by an attacker (*e.g.,* by physical keystore theft or by exploiting a crypto implementation bug [73]), the attacker may permanently release unauthorized updates. If the signing key is used on an app distributed on an application marketplace, the attacker would need to successfully gain access to the marketplace account to publish an update. Alternatively, the attacker could convince users to sideload the unauthorized version (*e.g.,* from a non-official site). Similarly with Baton, unauthorized updates will be possible if keys are compromised. This includes both standard updates as well as updates with a certificate chain. Developers using Baton must protect signing keys as usual. However, if key compromise or a crypto implementation bug is detected in a timely fashion, it may be possible for the legitimate developer to issue a Baton app update *before* the adversary, effectively "locking" users who upgraded into a new uncompromised replacement signing certificate.

6.6.4 Transferring Authority

Using Baton, developers can delegate signing authority to the holder of a different key, but the original certificate and corresponding key pair will remain authorized

for issuing updates to versions of the app not containing the certificate chain. For example, when an app is being sold, the seller may continue to issue updates to the app under the original key (see $V2_{SigA} \rightarrow V3_{SigA}$ in Figure 6.3). Clients who do not update to the Baton version ($V3_{SigB}$) may be tricked into installing updates with the old signing certificate instead. While there is generally a trust relationship established when ownership of an app is being transferred (*e.g.,* the buyer is already exposed to potential backdoors in the app), best practices would encourage revoking the original certificate from updating the app. This must be a finer grained revocation than certificate revocation: a seller of an app may have other apps signed with the same certificate that are not being sold. We consider two conditions—with and without a marketplace—under which apps could require a proper transfer of signing authority.

Assisted by a central marketplace. When an app is installed through an application marketplace, there are effectively two authentication mechanisms in place to ensure source continuity: the signature enforced at the OS-level and the developer account with which the app is associated at the market-level. Application marketplaces such as Google Play allow developers to transfer apps to another account [71], which effectively prevents an app seller from continuing to issue updates through the marketplace. For marketplace users, app updates will proceed as usual. Users who install apps from multiple markets or by sideloading may still be vulnerable to installing unauthorized updates that are signed with the original developer's key.

Without a central marketplace. When users install apps from only side-loaded sources, it seems difficult to communicate the revocation of a certificate's signing authority over a specific app. Certificate revocation remains a open problem in self-signed environments, where no single entity is authoritative except perhaps the OS itself.

Detection instead of prevention.

It could prove advantageous to keep a public record of package names and associated certificate chains as a type of public notary to identify if different certificate chains

emerge for the same app. This principle can be seen in other domains: *e.g.,* Perspectives [123] and Certificate Transparency [81] which aim to detect fraudulent certificates in TLS. A similar system could be used in Android to confirm the uniqueness of a certificate chain at install-time. Baton could be augmented to submit certificate chains or query valid chains for a given application by leveraging an install-time server query mechanism like Meteor, described in Chapter 4. The server-side component, which reports back on valid or invalid chains, would require manual curation by experts.

6.6.5 Applicability Beyond Android

The Baton protocol (see Protocol 1) is designed to be generically applicable in other decentralized signing environments. We only require that signed objects exist in a collision-free namespace. That is, the underlying OS prevents the existence of more than one signed object with the same name. In the case of Android, we use package names as identifiers. However, Linux file system paths could also be used, resulting in a direct and practical improvement over the key-locking proposal of van Oorschot and Wurster [117]. Baton also requires a way to keep track of versions to ensure correct validation of delegations. Object versioning can be implemented at the application level similar to Android's version code, or built in to the file system itself.

6.6.6 Limitations of Baton

One of Baton's main limitations is the need for developers to include the certificate chain (which includes corresponding full certificates) in potentially all[8] subsequent versions after a certificate transition. Failure to include the chain of certificates would prevent users who have not yet upgraded to the latest version from seamlessly upgrading, since there is otherwise no easy way to verify the chain. Android certificates are typically 600 bytes to 2 kilobytes, so overall app size is not expected to be adversely impacted by including several certificates. Since certificates and certificate chains are

[8]The certificate chain should be included in all subsequent versions from which the developer wishes to allow transparent upgrades, or as long as there is reason to believe not all users have performed the most recent certificate transition.

intended to be public, backup copies may without risk be stored in the cloud or on a shared drive.

Private key loss, even with Baton, remains a difficult problem. Losing a signing key means it is no longer possible to issue a Baton certificate update, unless a signature threshold system [109] is used. We believe solving this limitation would weaken Android's overall security model since a mechanism to issue an update without the original key could be abused by an adversary.

6.7 Alternative Approaches

In this section, we consider three alternatives to improving the decentralized signing model. As is common in PKI design, each alternative presents a set of trade-offs relative as opposed to an unequivocal improvement compared to our Baton proposal.

6.7.1 Centralized Private Key Infrastructure

One alternative to the decentralized model is to use a full-fledged centralized PKI where developers prove their identity to a certificate authority (CA) and are issued a certificate. In this model, the certificate information can link developers together as being the same person or from the same team without needing to use the exact same key to sign every app. While this model has beneficial security properties, the added complexity makes a centralized PKI a poor trade-off in our opinion. It would require developers to obtain CA certificates and Android to decide on the set of trustworthy CAs as well as make security decisions (or require the user to) based on certificate attributes. Being a heavy-weight alternative for limited benefit, and one that does not work particularly well in other domains (e.g., SSL/TLS), we do not recommend a central PKI as an improvement to the Android ecosystem.

6.7.2 Certificate Trees

The self-signed model can be combined with a constrained certificate hierarchy we refer to as a one-level certificate tree. A development team (or single developer) would establish a self-signed long-term CA certificate as the root of a one-level tree. The signing key would be well-protected and used only infrequently to issue shorter term certificates for actually signing apps. The update process would still use the leaf certificate in the tree to decide on allowing an update to an existing app but the root certificate could be used to establish a shared reputation amongst all certificates it issues. In a team environment, this could be a shared reputation across distinct developer certificates. With a single developer, it could be a shared reputation amongst distinct app certificates.

A certificate tree can reduce the need to replicate copies of the same key. It can also be useful if a developer wants to transfer ownership of a signed and widely installed app to another developer. If the signing key for an app is transferred to the buyer, the buyer can issue updates to all apps signed with the key. A certificate tree can allow individual signing keys for distinct apps while still allowing the developer to maintain a reputable link to their other apps. However, this is not a fully satisfactory solution as transferring a key does not sever the link to the CA certificate.

Certificate trees would require no changes to Android but redefine the purpose of a CA. Typically, the identity of the holder of a CA-issued certificate is believed to have been validated by the CA. Here, CA-issued certificates are used to collect several signing keys under one shared reputation. For the latter type of CA to distinguish itself from the former, they could use a naming convention that would be obvious (or, more formally, use an X.509 extended attribute).

6.7.3 Distributed and Threshold Signatures

An approach to protecting against the theft of signing keys is to use distributed signing, where n participants individually sign the app and all n participants are required to sign each update (*cf.* TUF update framework [106]). Android currently supports distributed signing. For each signature, an individual signature file (*e.g.*, `KEY1.SF`) is created and signed, appending the signature to individual certificate files

(*e.g.,* `KEY1.RSA`). Android will allow updates only if the same exact set of signatures are used with each update. Of all the apps in our dataset, only 42 had more than one signature. For 30 of these, the additional certificate was a temporary (and expired) debug certificate created by the Android app development tool for testing purposes. For 4 of these, a publicly available key pair was used—see Section 5 for discussion of this case. This leaves a small set of apps that appear to be using distributed signing (in each case, a 2-out-of-2 access structure) for security purposes. One example is the Mint app (`com.mint`), a financial management service.

In the current model, there is no cross-certification between the multiple signatures. This means signatures can be selectively stripped in distributed signing instances [117]. We verify that this is the case even if the first signatures applied are included in the signature file for subsequent signatures (*e.g.,* `KEY2.SF` includes `KEY1.SF` and `KEY1.RSA`). In one app from Sprint, (ostensibly) HTC added a signature that updated `MANIFEST.MF` with Sprint's `SF` and `RSA` files.[9] We stripped Sprint's signature and verified that the validation tool ignores files recorded in `MANIFEST.MF` that are located in the `META-INF/` subdirectory.

Android could be modified to allow apps to specify in `AndroidManifest.xml` (a file that is explicitly signed) the intention to use multiple signatures signing and pin the set of public keys required to update the app (*cf.* certificate pinning[10]). A simple extensions is to also permit developers to specify a t-out-of-n access structure that requires the signatures of any $t \leq n$ of a set of n signatures. This maintains the security of distributed signing while adding robustness against lost or unavailable keys [57].

For completeness, we note that threshold (and distributed) signing can be conducted externally to `jarsigner`. Threshold signing protocols are known for RSA signatures, both with a trusted dealer that generates the key shares [109] and with distributed key generation [37]. To the best of our knowledge, no tool is available that implements a threshold signature scheme and easily allows developers to perform such a protocol.

[9]This changes the hash of `MANIFEST.MF` from the one reported in Sprint's `SF` file, which the verification tool ignored. It appears to only verify the `MANIFEST` entries themselves.

[10]IETF Internet-Draft: HTTP Strict Transport Security (HSTS)

6.8 Summary

The analysis of real-world examples and high-profile applications clearly illustrates the need for a mechanism to allow changing signing keys and certificates associated with Android apps. We have demonstrated its viability by providing a practical instantiation which has been tested and shown to be compatible with the current Android ecosystem.

Baton demonstrates that what are typically considered as academic best-practices for certificate update and package signing can be moved from theory to practice, to create a practical and lightweight mechanism that establishes cryptographically verifiable trust chains between certificates. With Baton, the responsibility of verifying integrity and authenticity of updates is placed on the developer and the OS, lightening the load on the user. While we have implemented and evaluated Baton under the constraints of the Android ecosystem, we believe the overall Baton protocol will be of interest to other decentralized environments as a general mechanism to delegate trust between signing certificates.

Chapter 7

Applying and Enforcing Application Security Policies

7.1 Introduction

Throughout this thesis, we have shown that decentralized environments allow (and sometimes implicitly encourage) the installation of many applications from a broad range of developers. While users may enjoy the freedom afforded by these types of environments, the absence of a central authority places responsibility of security hygiene on the software installation framework, the OS and ultimately on users themselves.

In centralized ecosystems, an authority is in charge of vetting applications for inclusion in repositories and may even be authorized to remotely uninstall malicious applications from user devices. Since security is provided at multiple levels, security guarantees provided by the OS may not need to be as strong. Operating systems in decentralized environments, on the other hand, attempt to deliver on security expectations through the use of on-device application security policies which constrain the privileges of installed applications and limit access to user data.

The previous chapters have focused on identifying legitimate applications and preventing malicious updates. In this chapter, we discuss the final step in the software installation process in which an application security policy is retrieved and applied to the application being installed. We begin with a review of general mechanisms used to

accomplish this goal in decentralized environments. Next, we focus our discussion on Android's policies for application integration and note that several limitations exist in these policies. Application integration on Android is tied to digital signatures used on packages, thus limiting the number of other applications with which an application can be integrated. This model may also encourage bad practices such as sharing private signing keys or code with other developers and key reuse. To address these limitations, we propose Fluid, a flexible and secure policy to enable sharing privileges between applications signed with different keys. We use empirical data obtained from the Android Observatory (see Chapter 5) to inform our design such that Fluid can be deployed with minimal impact to the current application ecosystem.

7.2 Security Policies in Decentralized Environments

Application security policies describe a set of privileges that are granted or denied to a specific application once it is installed [107]. These application policies are designed to provide security guarantees according to the principle of least privilege [103] for stand-alone applications as well as define parameters for inter-application communication. Depending on the software installation model (see Section 2.3), policies may be written by a number different entities, as described below.

7.2.1 Application developers

Developers may provide a policy defining an upper bound on privileges needed by the application. The policy may be read by users or software to provide information regarding the level of privilege that the application will be granted once installed. Developers typically know what privileges are needed by their applications since they are ultimately the authors. However, application over-privilege (*i.e.,* requesting permissions beyond the set needed to provide expected functionality) becomes possible due to complexity in writing policies, developer apathy towards security, or incompetence [51].

On Android, developers must supply a policy in the form of a list of permissions

that the application requires to function. The list of permissions requires manual specification from developers, who are free to request as many permissions as they want, even permissions not actually required for regular application use. On Android, an implicit *deny* clause is added to the application's security policy for all privileges that are not explicitly requested.

7.2.2 Guardians

Guardians (recall Section 2.3.3) have a vested interested in the security of system, and therefore are concerned with constraining applications that are installed by the users they protect. Guardians may create policies based on the expected functionality of an application. For example, if an application is believed to take input from the command line, process the input and display output, a policy could be written denying the application access to network sockets and file I/O. If the application was found to have an arbitrary code execution vulnerability, this policy would prevent the application from communicating via the network or reading and writing to local files.

A practical example illustrating guardian-written policies is the AppArmor [12] project. AppArmor is a mandatory access control (MAC) framework for Linux that allows custom policies for userspace applications. AppArmor is widely deployed (default MAC mechanism on Ubuntu and openSUSE, and available on most Linux distributions as of 2014), and is highly configurable. In distributions where AppArmor is enabled by default, the OS vendor acts as the guardian creating policies for many privileged applications. System daemons such as `ntpd` and `cupsd`, and privileged applications such as `tcpdump` are confined by AppArmor profiles allowing only necessary privileged operations. Userspace applications installed outside of the distribution's package manager are not protected with AppArmor unless a policy is explicitly supplied by the application's developer or user.

Another, less widely deployed example of guardian-written policies is found in the SELinux project [84]. SELinux is the implementation of a MAC framework for Linux, as is enabled by default in Fedora [50]. As in AppArmor, Fedora developers are the guardians who maintain a set of policies confining applications that ship as part of the default install. Third-party userspace applications are left by default in the

`unconfined` domain, which gives applications most privileges. As of writing, Fedora is experimenting with creating policies to restrict additional privileges for applications in the `unconfined` domain.

7.2.3 Users

Users are often given the option to configure their systems, especially in decentralized environments. While users are generally free to rewrite application policies (*e.g.,* by modifying an AppArmor profile), we expect most users to lack sufficient technical knowledge to do so. Thus, security policies tend to be configured through user-friendly system or application settings screens (often allowing only the modification a few policy entries). These settings will allow users to further remove privileges from installed applications, but generally will not allow the user to grant *additional* privileges. Granting additional privileges can be accomplished, however, by disabling the policy enforcement mechanism (*e.g.,* setting SELinux or AppArmor to permissive mode).

In Figure 7.1, we show a third-party Android build (CyanogenMod [36]) which includes a tool allowing users to deny individual permissions on a per-application basis. The tool displays the list of permissions which were granted to the application at install-time, including the number of times API calls protected by the permission have been used by the application. Through this interface, users can configure permissions to be *allowed*, *denied*, or *always ask*. There is no mechanism to add new permissions to the list. Blackberry OS and Apple iOS have supported a similar user-configured policy for a subset of privileges assigned to applications.

7.2.4 OS Vendors

OS vendors may also create security policies for applications. In centralized environments, the central authority may have the capacity to create and distribute (either through a separate channel or bundled within an application) policies for each third-party application that is made available to users. This is possible because (1) all available applications are known to the authority; and (2) all applications must be

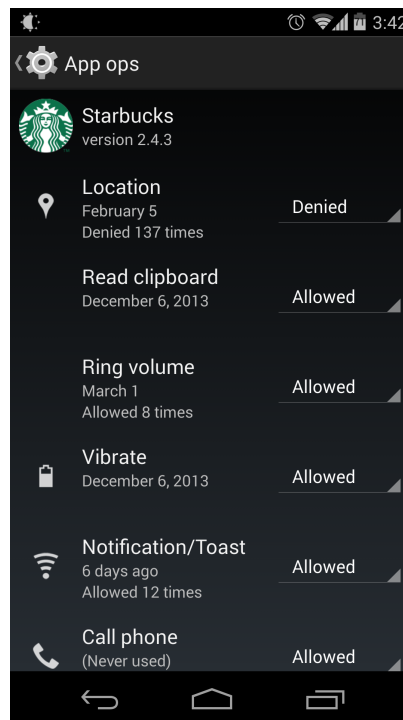


Figure 7.1: Screenshot of the AppOps tool on a CyanogenMod Android build. The tool displays the permissions granted to the application (including time and date of last access to a permission), and allows users to individually deny permissions for the specified application.

submitted to the authority for inclusion in an application marketplace. Decentralized environments cannot know all possible applications that exist for the platform, and thus OS vendors focus their application policies on only system-provided applications. The SEAndroid [110] project maintains a set of SELinux policies for privileged Android applications that ship as part of the base OS image such as the volume mounter (`vold`), and the Android *init* process (`zygote`).

It may be tempting to think that OS vendors should write policies for third-party applications in decentralized environments. However, without prior knowledge about the behaviour of all possible third-party applications, these policies would need to be permissive enough that malicious behaviour would ultimately be allowed. Otherwise, benign applications may be inadvertently constrained and be unable to provide their expected functionality. Similar to SEAndroid, on Samsung devices the Knox [104] MAC framework includes policies to help prevent root access by enforcing security policies on system applications. No guarantees beyond Android's stock security model (see Section 3.2) are provided regarding the security of third-party applications since security policies are not provided for these applications.

7.3 Android Policies for UID Sharing

In Android's current security model, new apps are assigned unique UIDs to enforce filesystem and process isolation, as well as grant access to privileged device capabilities (see Section 3.6). As previously discussed, UIDs are an essential component of the Android security model. Android allows developers to write apps that share a UID. To use this feature, developers set the `sharedUserId` directive to a common string value in the `AndroidManifest.xml` file of each app that will share the UID. The apps must be signed with the same key (or set of keys) for Android to allow a shared UID. The common string allows, for example, two apps signed with the same key to each belong to a different UID sharing group (and thus have a distinct UID).

While Android already allows developers to write modular apps that integrate with each other via interprocess communication (IPC) [46], UID sharing allows developers to split app functionality into multiple installable components, yet share binary resources such as fonts, images and sound clips. Android core applications installed

SharedUID String	Distinct apps	Description
`mgeek.dolphin.browser`	41	Dolphin browser and extensions (*e.g.,* PDF viewer)
`com.doodlemobile`	22	Miscellaneous games
`com.handcent.sms`	14	Handcent SMS application and plugins (*e.g.,* fonts, emoji)
`com.jb.gosms`	10	GO SMS Pro application and plugins (*e.g.,* chat, support, Dropbox)
`com.swn.netgallery`	7	Wallpaper browser and wallpaper packs
`com.mxtech`	6	MX video player and related codecs
`com.jb.gokeyboard.shared`	6	GO Keyboard and language packs
`android.uid.system`	6	Miscellaneous applications installed as part of the base OS image
`org.jraf.android`	5	Live wallpaper viewer and wallpaper packs
`com.wsl.CardioTrainer`	4	Miscellaneous fitness applications

Table 7.1: Top 10 most-used shared UID strings on the Android Observatory. For each sharedUID, we list the number of distinct package names that use the stated UID string, as well as a brief description of the applications that use the sharedUID.

in the base image use shared UIDs to interact with other system components. For example, the `sharedUserId` "`android.uid.system`" is used by the Settings, VPN, and AccountsAndSync settings in stock Android apps. These three applications share a single logical sandbox, where privileges granted to any application can be used by all applications in the group.

In a snapshot from April 15, 2014 of the Android Observatory (see Section 5) we observed 258 distinct shared UID strings across all applications. These strings were used by 426 distinct package names. However, of these, we observed 206 instances of apps that declared a shared UID, but no other app in our dataset used the same string (this is not surprising, as our dataset is still only a small subset of all Android apps). For the remaining 52 shared UID strings, we found 226 distinct package names, leading to a 4.34 apps/sharedUID ratio. In total, 874 binaries (in some cases multiple versions of the same application) in our dataset used or attempt to use the shared UID feature. One shared UID string was declared by 41 distinct package names (`mgeek.dolphin.browser`). Table 7.1 lists the top 10 most used shared UID strings.

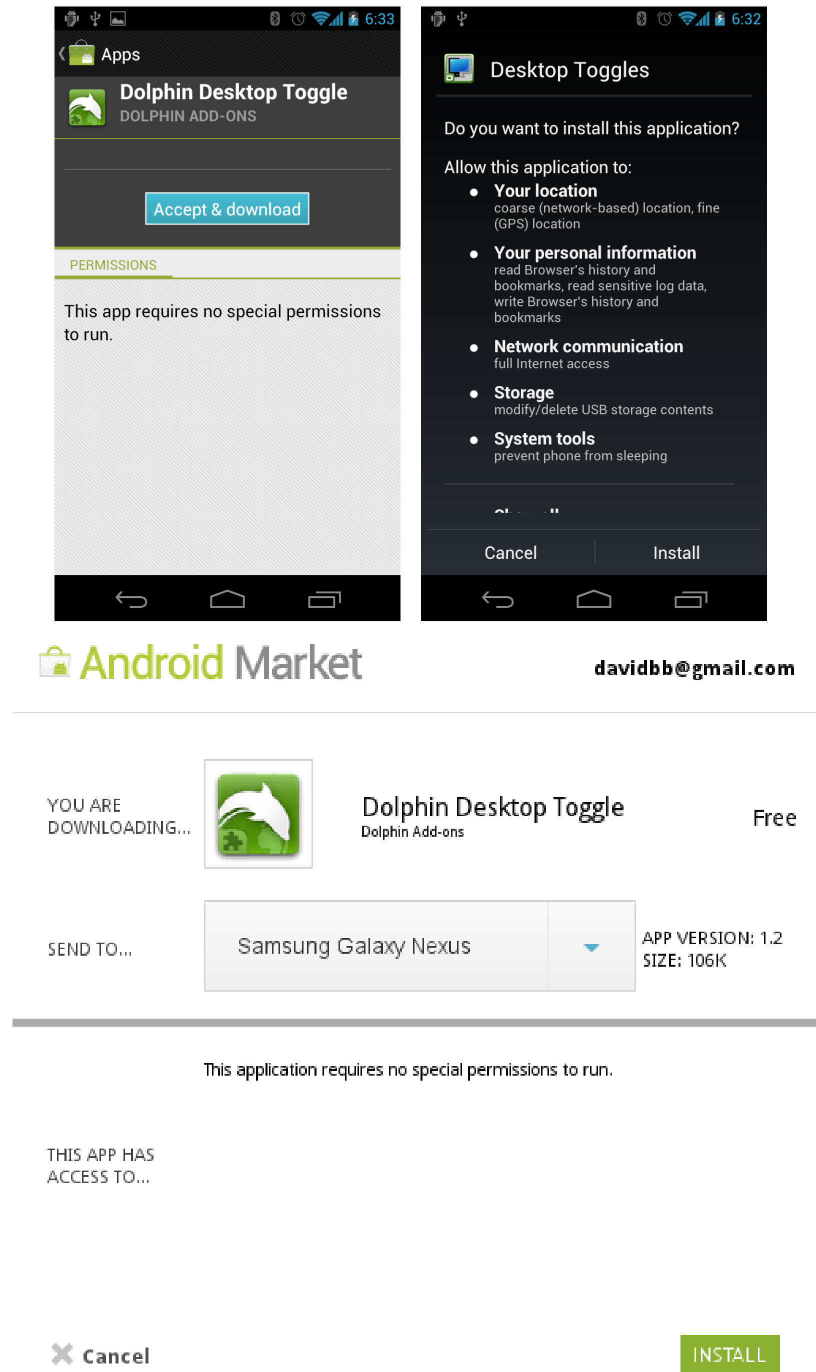


Figure 7.2: Screenshots displaying permissions requested by installing the Desktop Toggle browser extension via the Google Play Store (top left), via sideload (top right), and via the web (bottom). In all cases, the Dolphin Browser had been previously installed.

7.3.1 Permission Inheritance through UID Sharing

Through its user interface, Android implicitly treats permissions as if they were assigned to apps. However, as described in Section 3.4, permissions are assigned to UIDs. This distinction is generally inconsequential, except in the case when applications share a UID. Upon installation of a new app which will share a UID with an existing app, the new app will be granted all the permissions that have been granted to the existing app in addition to its own requested permissions. We call this effect *permission inheritance*, where a new application inherits permissions that already exist in the UID group. Likewise, the existing app will be granted all the permissions associated with the new app. We call this *retroactive permission inheritance*.

An example of permission inheritance is shown in Figure 7.2. Here, we show device screenshots taken during the installation of a browser extension which requests no permissions in its manifest, but shares a UID with a highly privileged browser. When the extension is installed from the official Google Play Store[1], the installation framework states that "this app requires no special permissions to run". However, due to permission inheritance, the extension is granted the same permissions set as the browser. The correct permission set is only displayed when the extension is side-loaded. We note as well that if the extension is installed through the Google Play web interface, a similar permission-less screen is displayed.

Permission inheritance inconsistencies occur when the application installer does not have an accurate, updated view of all installed apps on a device. Google keeps track of applications installed through the Google Play store, but does not know about side-loaded apps. If Google knew about all apps installed (through the Play Store and side-loaded), the Google Play app should be able to accurately display the effective resulting permission set at install-time. The package installer invoked when side-loading does have an accurate list of all installed apps, and can therefore correctly display the permission list.

One possible solution to this problem is to always use the system-provided application installation framework when installing apps that have the `sharedUserId` attribute set in their manifest. The user experience in these cases may suffer, since users would

[1] This test was performed using the Google Play Store app version 3.5.16 on Android 4.0.4, but this behaviour is also reproducible on earlier versions of Android.

see an unfamiliar (*i.e.,* different than that of Google Play) installation screen. The application installation framework could additionally be augmented to display which permissions are requested by each app in the shared UID group, rather than only display the union of all permissions.

7.4 Improving Android's UID Sharing Policies

The main limitation of UID sharing as currently implemented by Android is its inherent dependency on app signing, which precludes apps signed with different private keys from sharing a UID. First, this policy conflicts with the reputation-based model implicitly encouraged by Android's app signing protocol (see Section 3.5). In this model, developers re-use the same private key across multiple apps to leverage reputation gained from the development of other apps, but sharing a UID with an application signed with a different key becomes impossible. Second, it encourages developers to pursue application integration through custom mechanisms, such as IPC which, by default, offers no authentication and is more open to developer error.

While developer-defined permissions can protect access to an API, the developer then has to choose between either (1) granting every app the permission or (2) granting only apps signed with the same key the permission (see Section 3.6). This dichotomy is a direct result of permissions in Android being assigned to UIDs, and not applications. We would like to retain the authentication properties afforded by digital signatures to prevent untrusted applications to share a sandbox, but our goal is to create flexible policy that allows authenticated applications to join UIDs, while excluding untrusted applications.

As shown in Table 7.1, UID sharing is commonly used in situations where small portions of functionality are split into multiple installable apps. One example of UID sharing observed in our dataset was a web browser that shared its UID with browser extensions. In this example, it is natural to consider that the extension may be authored by a different developer than the browser, and yet the browser's developer must sign the extension for the extension to function correctly. This could be problematic for the browser developer as code signing is typically considered as an implicit endorsement of someone else's code, and it also fails to allow the extension developer to

preserve any reputation from other apps they have released. The extension developer also loses attributed authorship.

We now consider alternatives for improving UID sharing by addressing the limitations above.

Properties for UID Sharing

Android's current model for UID sharing provides a number of important properties which we have distilled into Properties 1 to 6 below (we denote apps that share a UID as members of a *group*):

P1. **Groups are Disjoint:** Linux, and therefore Android processes, run under a single UID. Thus, there is no way for an instance of a running app to be a member of two or more groups.

P2. **Groups are Consistent:** If app A is in the same group as B, and app A is in the same group as C, then B and C must be in the same group.

P3. **Group Size is Arbitrary:** It is possible to specify UID-sharing groups of size 1, 2 or greater than 2.

P4. **Membership is Authorized:** An app cannot join a group without being authorized[2] to join.

P5. **Adding New Members is Efficient:** If a new member is added to a group, ideally no other group members require an update. In the next best case, only one app requires update (denoted with ∘ in Table 7.2). In the worst case, all members of the group require update.

P6. **Different Groups with Same Key:** It is possible for apps signed with the same key to be members of different groups (while preserving the properties above).

We seek to design a UID sharing mechanism that would satisfy all 6 properties above as well as enable an additional property:

[2] Authorization on stock Android is expressed by signing all the apps in a group. Thus, it should be impossible to join a group without access to the group's private key.

	Mutual Approval	MaxTwo	Fluid	Signature-based
1. Groups are Disjoint	•	•	•	•
2. Groups are Consistent		•	•	•
3. Group Size is Arbitrary	•		•	•
4. Membership is Authorized	•	•	•	•
5. Adding New Member is Efficient†		◦	◦	•
6. Different Groups with Same Key	•	•	•	•
7. Different Keys within Same Group	•	•	•	
Viable			•	•

Table 7.2: Comparison of proposed UID sharing mechanisms to signature-based sharing (the current model). See inline discussion for explanation of "viable".
† For new applications added to the group, • means only one member must be updated, ◦ means two members must be updated, and empty means all members must be updated.

P7. **Different keys within the same group** Applications in a UID sharing group may be signed with different private keys.

However, in the course of redesigning the mechanism, we found that in every case, adding property 7 precludes at least one other existing property. We therefore consider the relative merits of trading off various properties in order to achieve property 7.

7.4.1 Alternative Mechanisms for UID Sharing

We consider three possible alternatives to Android's currently implemented mechanism for UID sharing: mutual approval, pairs, and our own proposal, Fluid (see Table 7.2 for a comparison matrix).

- **Mutual Approval:** Each app specifies in its manifest every other app in its UID sharing group. Apps in the group are specified by package name and fingerprint of their embedded signing certificate. This fulfills the goal of allowing apps signed by different keys to share a UID, but makes it difficult to add new members to the group as every app in the group must be updated (by having

each developer create a new application release that specifies the newly added group members) for each new member addition. Mutual approval is also prone to group inconsistencies since it is possible that one app (A) approve two apps unilaterally (*e.g.,* B and C), without B and C mutually approving each other. Android has no clear mechanism to resolve such inconsistencies, so we conclude mutual approval is not a viable alternative to the current signature-based model.

- **MaxTwo:** This alternative constrains groups to a maximum size of two and can be seen a sub-case of the Mutual Approval mechanism above. This restriction helps ensure group consistency, but has the obvious disadvantage of not allowing groups larger than two apps. In our dataset, many instances of apps sharing a UID were larger than two, so we do not consider pairs to be a viable alternative.
- **Fluid:** In this alternative, one main app in each potential group of apps (*e.g.,* a browser) authorizes in its manifest all other apps (*e.g.,* plugins, language packs, *etc.*) in the group. This resolves the group inconsistencies of mutual approval as well as meeting our original goal of separating UID sharing from application signing. The main drawback of Fluid, as we discuss further below, is that the main app must be updated every time a new group member is added.

7.4.2 Implementing "Fluid" UID Sharing

To implement the Fluid UID sharing mechanism, we made changes to the core Android package installer framework classes (`PackageManager.java` and `PackageManagerService.jav`
Since we are modifying part of the base system image, we also add a new set of XML attributes to the official API. This modification has the downside of requiring Google (or a popular third-party Android build) to incorporate our Fluid modifications for wide-scale deployment (*i.e.,* Fluid is not installable as an app, but rather part of the base OS installation framework). Google routinely updates their OS and manifest specification. For example, in API level 3 (Android version 1.5), Google introduced the `sharedUserLabel` directive [68].

We suggest adding the following XML attributes to the Android Manifest: `sharedUserIdMode,` `nodeRoot`, and `nodeLeaf`. The `sharedUserIdMode` attribute specifies if the application in question is intended to be a root (as in tree, not as in user) application, on

which plugins and extensions are built; a leaf that provides an additional service to a root application; or to use the pre-existing signature-based UID scheme. Not specifying a `sharedUserIdMode` defaults to the pre-existing scheme for backwards compatibility with apps that are no longer updated. The `nodeLeaf` attribute specifies (for the main application) each of the plugins or extensions with which it should share a UID. Plugin applications are specified via package name and certificate fingerprint (see Figure 7.3). Finally the `nodeRoot` attribute specifies the package name and certificate fingerprint of the main app with which to share a UID (see Figure 7.4).

We note that with our modifications, some new restrictions must be enforced by Android at install-time to guarantee integrity and consistency.

- An application must not declare both `nodeLeaf` and `nodeRoot`.
- Applications must only specify one of "root", "leaf" or "signature" for the `sharedUserIdMode` attribute.

In Fluid, shared UID labels are not required as in the traditional signature-based scheme, since both leaf apps and root apps specify other apps in their manifest. At install-time, after verifying the restrictions listed above, the installer looks for apps specified as `nodeRoot` or `nodeLeaf` in the newly installed app's manifest. If matching apps are found and manifests are consistent (*i.e.,* the root and the leaf specify each other), leaf apps are updated to use the root application's UID.

7.4.3 Discussion

Mutual Authentication

Fluid requires authentication (by explicitly specifying the package name and certificate fingerprint of other apps in the group) from the main application as well as from leaf applications to allow sharing UIDs. This prevents applications from arbitrarily joining UIDs when they are not authorized to do so (a desirable property). Fluid also removes the code signing requirement for sharing UIDs. Thus, developers can use different signing keys and certificates on each application they author, following best

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 ...
        package="com.cool.browser"
        android:versionCode="12"
        android:versionName="1.0.1"
        android:sharedUserIdMode="root"
        android:nodeLeaf="com.supercool.extension1, 67709F9D63BFD6A0"
        android:nodeLeaf="com.supercool.extension2, 774A0E232918EE8B"
        android:nodeLeaf="com.cool.extension, 0DCDD39F12A07A25" >
...
</manifest>
```

Figure 7.3: Example Android manifest of a web browser, acting as the root application for 3 extensions. Note that all extensions are signed by a different key (denoted by different fingerprints on the right). The first two extensions are written by the same development company (but signed with different keys), while the third is written by the browser vendor.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 ...
        package="com.supercool.extension1"
        android:versionCode="3"
        android:versionName=".98"
        android:sharedUserIdMode="leaf"
        android:nodeRoot="com.cool.browser, 0DCDD39F12A07A25" >
...
</manifest>
```

Figure 7.4: Example Android manifest of a browser extension using Fluid. The developer specifies that the extension should share a UID with the browser.

practices. If developers sell one or more components in a UID group to a different developer, a technique such as Baton could be used (see below).

Evicting Applications from a UID Group

Fluid gives developers control over which other applications to share UIDs with. We consider the case of a browser extension developer, previously authorized via Fluid to share a UID with a browser, going rogue (*e.g.,* changing development from benign to malicious). When malicious activity is detected by the browser developer (*e.g.,* through user or crash reports), the developer can issue an update removing the malicious extension from the manifest. Upon installing the update, the malicious extension would be removed from the group, but remain installed on the device.[3]

Baton Integration

Fluid could be adapted to work in the event of Baton-style certificate delegations (see Chapter 6). Fluid requires developers to specify the fingerprint of the *currently active* certificate in the Fluid section of the manifest. If certificates change, Baton will not "kick out" applications from the group (see Section 6.5.1), allowing that behaviour to be decided by the developers themselves. Fluid could be modified to look for either the currently valid certificate, or a once-valid certificate in the certificate delegation chain. This would allow shared UID leaf applications to continue functioning (without changes or updates) even if the root application updated its signing certificate.

Incremental Deployment

To encourage adoption of Fluid, and enable incremental deployment, we preserve traditional UID sharing functionality. However, as soon as one application in a UID group enables Fluid UID sharing, all applications in the group must be updated as well. If instead we had chosen to allow only root or only leaf applications to enable Fluid, this could have resulted in arbitrary applications being assigned unauthorized

[3]Android has no built-in mechanism to uninstall a different application when updating an application. Thus, the evicted application would remain installed. However, since the recently evicted application no longer belongs to the previous UID group, it would no longer have access to the privileges granted to that group.

UIDs. This is regarded as a security risk, so we chose to disallow this functionality in our design.

7.5 Related Work

Ontang *et al.* [97] propose the use of fine grained policies governing application interaction on Android. Their proposal, Saint (Secure Application INTeraction) allows the application developer to specify detailed rules for controlling access to exposed application interfaces. For example, an application can make available an API to other applications only if they satisfy a certain property that can be checked at runtime (*e.g.,* applications invoking the API must not have a blacklisted package name). The Saint proposal focuses on IPC, and appears (though not explicitly) to not modify or address UID sharing policies on Android. The Fluid architecture could be used in conjunction with Saint, enabling app interaction policies for both IPC and UID sharing.

Xu *et al.* [129] propose re-packaging applications to include custom runtime libraries and security policy. This allows end-users to specify their own policies (*e.g.,* this application should not be granted network access). The repackaged applications should therefore have lower privileges. One advantage of this proposal is that the OS requires no modifications, since all changes are made to the applications themselves. However, since applications must be re-packaged and re-signed, original authorship information is lost, which might break UID sharing if enabled. Fluid could be used to augment the proposal of Xu *et al.* by repackaging all applications in the UID sharing group, and adding the correct entries to each of the group member's manifests.

Xing *et al.* [128] identified a security risk in the way that OS updates enable new functionality. Applications may request permissions to access features and functionality not available in an old version of Android, which the installer does not display to users. If the app remains installed, the OS is updated, and the new features become available, privileges will be granted automatically to previously installed applications. This behaviour appears to be similar in nature to our permission inheritance attacks (see Section 7.3.1), but require the updating of the full OS instead of only an application update.

7.6 Summary

This chapter has reviewed different mechanisms to deliver application security policies to devices as part of the installation process. We have focused on application security policies in decentralized environments, which by definition cannot use the tools available in centralized environments.

We have discussed issues pertaining to Android's implementation of UID sharing privileges which at present is tied to digital signatures. Digital signatures provide strong authentication guarantees, but prevent sharing privileges with applications authored by other developers, unless private signing keys are shared. We proposed Fluid as an alternative UID sharing scheme, which allows developers to specify which other applications should share a UID using a flexible policy language. Based on the analysis of over 50,000 applications collected from the Android Observatory, Fluid is compatible with current practices, and should be incrementally deployable.

Chapter 8

Discussion and Conclusions

8.1 Revisiting Thesis Goals

In Section 1.1, we stated two overall goals for this thesis. We revisit these goals, discussing how the contributions herein are in line with our objectives.

G1. Improve Decentralized Software Installation Security Mechanisms.

Our first major goal was to identify limitations in existing security mechanisms used in decentralized environments, and to address these limitations without requiring significant architectural redesigns. Where possible, we wish to minimize negatively impacting deployed applications, as well as enable incremental deployment and backwards compatibility. In this thesis, we proposed two practical improvements to existing security mechanisms used in real-world decentralized ecosystems.

First, in Chapter 6, we proposed Baton, a protocol that allows developers to update or change the signing certificates used on their applications. Baton solves long-term certificate lock-in in decentralized environments, which ultimately facilitates and supports the use cryptographic best-practices. Baton builds on existing academic work, and has been implemented and tested as a proof-of-concept on Android. The Baton protocol is proposed to be a generic mechanism applicable to other decentralized environments where digital signatures are used to authenticate installation packages.

In Chapter 7, we discuss limitations in Android's UID sharing design arising from its dependence on digital signatures. To address these limitations, we propose Fluid, a mechanism for specifying application integration policies based on mutual authorization of applications. Fluid allows developers to integrate (*i.e.,* share privileges among) applications signed with different keys, and provides an effective mechanism to remove established integration between apps as needed.

G2. Propose New Architectures to Enhance the Security of Decentralized Software Installation.

Our second major goal was to propose novel architectures and tools that could be incompatible with current ecosystems, but where the security benefits outweigh the deployment challenges. With this objective in mind, in Chapter 4 we identify the key differences between centralized and decentralized security installation and propose Meteor as an alternative software discovery and pre-installation architecture. Meteor leverages domain-specific expert crowd-sourced information to help end-users obtain as much information as possible prior to installing an application. We propose the use of decentralized kill switches to regain some of the security properties lost by moving from a centralized to a decentralized structure.

In Chapter 5, we describe the design and implementation of the Android Observatory. We have made the Android Observatory a publicly available web service that indexes Android application metadata. It allows users (including non-experts) to quickly display permissions, signing information, and other package properties by searching or uploading applications. The Observatory has served as an invaluable tool used to inform and motivate many proposals in this thesis, and has assisted other researchers in areas beyond our current scope.

8.2 Open Problems in Decentralized Software Installation

While the collection of improvements to decentralized architectures presented in this thesis have been demonstrated to offer concrete benefits, we now identify several remaining open problems in decentralized software installation. We note that these

open problems exist in similar domains where centralized PKIs are being replaced by decentralized alternatives (*e.g.,* SSL/TLS).

1. **Bootstrapping trust.** By definition, decentralized environments cannot rely on a central authority to provide identity assurances. However, once the initial trust between users and developers has been established, there are mechanisms to preserve that trust across updates. The problem, then, lies in the first installation of an application, which bootstraps the trust relationship. Other than creating tools to allow users to verify identities through multiple, distinct channels, it is unclear how to provide strong identity guarantees without a central authority. Furthermore, it remains a challenge to bootstrap trust with acceptable levels of both security and usability.

2. **Efficient trust revocation.** Similar to the point above, revoking trust (*e.g.,* in the case of private key theft or loss) is a known difficult problem in decentralized environments. The main existing proposals to address this problem are to use short-lived certificates and to disable trust if certificates expire. The Baton proposal enables the use of short-lived certificates (through allowing certificate agility), but offers no way to broadcast revocation of a certificate to all relying parties. We note that efficient revocation, including in centralized architectures, is a difficult problem.

8.3 Summary

In this thesis, we have shown that the properties of decentralized ecosystems present challenges in providing secure software installation and updates. We have distilled the software installation process breaking down and describing in detail each of its three main steps: discovery, download and updates, and application of security policies. At each step, we identified security limitations, and proposed improvements and new architectures to address those limitations. We have shown, through practical implementation and evaluation on a real-world OS, that it is possible to retain many of the security benefits of a centralized installation infrastructure through informed and careful design. We note, however, that there appears to be a necessary trade-off between freedom and convenience in decentralized environments. Certain users

will prefer the more restrictive but less involved aspects of centralized models, while others will gravitate toward fully decentralized ecosystems.

References

[1] ABI Research. Q4 2013 Smartphone OS Results: Is Google Losing Control of the Android Ecosystem? https://www.abiresearch.com/press/q4-2013-smartphone-os-results-is-google-losing-con, January 2014. [Online; accessed February 12, 2014].

[2] Adblock Plus. Surf the web without annoying ads. https://adblockplus.org. [Online; accessed March 13, 2014].

[3] Amazon. Amazon.com: Apps for Android. http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011, 2013. [Online; accessed September 30, 2013].

[4] Amazon.com. Customer Discussions: An Apology from Amazon. http://www.amazon.com/forum/kindle/ref=cm_cd_et_md_pl?_encoding=UTF8&cdForum=Fx1D7SY3BVSESG&cdMsgID=Mx2G7WLMRCU49NO&cdMsgNo=1&cdPage=1&cdSort=oldest&cdThread=Tx1FXQPSF67X1IU&displayType=tagsDetail#Mx2G7WLMRCU49NO, 2009. [Online; accessed March 19, 2014].

[5] J. Anderson, J. Bonneau, and F. Stajano. Inglorious Installers: Security in the Application Marketplace. In *WEIS*, 2010.

[6] R. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham. A new family of authentication protocols. *ACM SIGOPS Operating Systems Review*, 32(4), 1998.

[7] Android Developers. Android KitKat - Security. https://developer.android.com/about/versions/kitkat.html#44-security. [Online; accessed

February 10, 2014].

[8] Android Developers. Bytecode for the Dalvik VM. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. [Online; accessed February 10, 2014].

[9] AndroTotal. Scan Android Applications. http://andrototal.org, 2014. [Online; accessed April 14, 2014].

[10] AOSP. Issue 10020 - Can't renew signing cert. https://code.google.com/p/android/issues/detail?id=10020. [Online; accessed April 15, 2014].

[11] APK store. A store for all types of apps, games for your android device. http://www.apkstore.website.org/index.html. [Online; accessed March 14, 2014].

[12] AppArmor. Main Page. http://wiki.apparmor.net/index.php/Main_Page, 2014. [Online; accessed March 19, 2014].

[13] Apple. iOS App Store Review Guidelines. https://developer.apple.com/appstore/resources/approval/guidelines.html, 2011. [Online; accessed January 24, 2014].

[14] Aproov. App Store. http://www.aproov.com/. [Online; accessed July 23, 2012 (no longer available)].

[15] Archlinux. pacman - The Pacman Package Manager. https://wiki.archlinux.org/index.php/pacman, 2014. [Online; accessed January 23, 2014].

[16] J. Arkko and P. Nikander. Weak authentication: How to authenticate unknown principals without trusted parties. In *Security Protocols*, 2002.

[17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *ACM CCS*, 2012.

[18] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. NIST Special Publication 800-57 Recommendation for Key Management. 2012.

[19] E. Barker and A. Roginsky. NIST Special Publication 800-131A Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, 2011.

[20] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android. In *ACM SPSM*, pages 81–92, 2012.

[21] D. Barrera, H. Kayacik, P. C. van Oorschot, and A. Somayaji. A Methodology for Empirical Analysis of Permission-based Security Models and its Application to Android. In *ACM CCS*, pages 73–84, 2010.

[22] D. Barrera and P. C. van Oorschot. Secure Software Installation on Smartphones. *IEEE Security & Privacy*, 9(3):42–48, May 2011.

[23] A. Bellissimo, J. Burgess, and K. Fu. Secure Software Updates: Disappointments and New Challenges. In *HotSec*, pages 37–43, 2006.

[24] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical report, Technische Universitat Darmstadt, 2011.

[25] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS*, 2012.

[26] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android Categories and Subject Descriptors. In *ACM SPSM*, 2011.

[27] J. Cappos. *Stork: Secure Package Management for VM Environments*. Phd thesis, University of Arizona, 2008.

[28] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *ACM CCS*, 2008.

[29] P. H. Chia, A. Heiner, and N. Asokan. Use of Ratings from Personalized Communities for Trustworthy App Installation. In *Nordsec*, 2010.

[30] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this App Safe? A Large Scale Study on Application Permissions and Risk Signals. In *WWW*, 2012.

[31] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *ACM MobiSys*, 2011.

[32] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. *Journal of the ACM*, 45(6):965–981, 1998.

[33] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[34] J. Clark, C. Adams, and P. C. van Oorschot. Usability of anonymous web browsing: An examination of Tor interfaces and deployability. In *SOUPS*, 2007.

[35] Contagio Mobile. Mobile Malware Mini Dump. http://contagiominidump.blogspot.com. [Online; accessed March 14, 2014].

[36] CyanogenMod. Android Community Operating System. http://www.cyanogenmod.org/, 2014. [Online; accessed March 19, 2014].

[37] I. Damgard and M. Koprowski. Practical threshold RSA signatures without a trusted dealer. In *EUROCRYPT*, 2001.

[38] Debian. About Debian. http://www.debian.org/intro/about#what, 2014. [Online; accessed January 24 ,2014].

[39] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach. Quire: lightweight provenance for smart phone operating systems. In *USENIX Security*, 2011.

[40] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security*, 2004.

[41] A. Doan, R. Ramakrishnan, and A. Halevy. Crowdsourcing systems on the World-Wide Web. *Communications of the ACM*, 54(4):86–96, 2011.

[42] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, 2011.

[43] 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 *USENIX OSDI*, 2010.

[44] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security*, 2011.

[45] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *ACM CCS*, 2009.

[46] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security & Privacy*, 7(1):50–57, Jan. 2009.

[47] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. http://tools.ietf.org/html/draft-ietf-websec-key-pinning-11, Feb. 2014. [Online; accessed April 14, 2014].

[48] F-Droid. F-Droid. http://f-droid.org, 2013. [Online; accessed September 30, 2013].

[49] Federal Communications Commission. Letter to Apple regarding Google Voice and related iPhone applications. DA 09-1736, July 2009. http://hraunfoss.fcc.gov/edocs_public/attachmatch/DA-09-1736A1.pdf.

[50] Fedora Project. SELinux. https://fedoraproject.org/wiki/SELinux, 2014. [Online; accessed March 21, 2014].

[51] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS*, pages 627–638, 2011.

[52] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *ACM SPSM*, 2011.

[53] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *USENIX WebApps*, 2011.

[54] A. P. Felt, E. Ha, S. Egelman, and A. Haney. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[55] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.

[56] E. Gamma, R. Helm, R. Johnson, and J. Vlisides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[57] D. E. Geer Jr. and M. Yung. Split-and-delegate: Threshold cryptography for the masses. In *FC*, 2002.

[58] I. Goldberg, A. Mashatan, and D. Stinson. On message recognition protocols: Recoverability and explicit confirmation. *International Journal of Applied Cryptography*, 2(2), 2010.

[59] Google. Developer Tools - Android Developers. http://developer.android.com/tools/index.html. [Online; accessed April 15, 2014].

[60] Google. Google Cloud Messaging for Android - Android Developers. http://developer.android.com/google/gcm/index.html. [Online; accessed April 15, 2014].

[61] Google. Signing your Applications - Android Developers. http://developer.android.com/tools/publishing/app-signing.html. [Online; accessed April 14, 2014].

[62] Google. Upgrading to Google Authenticator v2.15 - Accounts Help. http://support.google.com/accounts/bin/answer.py?hl=en&topic=1099586&answer=2544996. [Online; accessed April 15, 2014].

[63] Google. An Update on Android Market Security. http://googlemobile.blogspot.ca/2011/03/update-on-android-market-security.html, March 2011. [Online; accessed March 4, 2014].

[64] Google. Google Bouncer - Official Google Mobile Blog . http://googlemobile.blogspot.ca/2012/02/android-and-security.html, 2012. [Online; accessed January 27, 2014].

[65] Google. Platform Versions - Android Developers. http://developer.android.com/resources/dashboard/platform-versions.html, Feb. 2012.

[66] Google. Android Apps on Google Play. https://play.google.com/store/apps, 2013. [Online; accessed September 30, 2013].

[67] Google. Android Security Overview — Android Open Source. http://source.android.com/devices/tech/security/index.html, 2013. [Online; accessed December 13, 2013].

[68] Google. <manifest> - Android Developers. http://developer.android.com/guide/topics/manifest/manifest-element.html, 2013. [Online; accessed March 18, 2014].

[69] Google. Manifest.permission - Android Developers. http://developer.android.com/reference/android/Manifest.permission.html, 2013. [Online; accessed December 18, 2013].

[70] Google. Android Open Source Project. http://source.android.com, 2014. [Online; accessed April 11, 2014].

[71] Google. Transfer your application - Android Developers Help. https://support.google.com/googleplay/android-developer/checklist/3294213?hl=en&ref_topic=3450986, 2014. [Online; accessed April 15, 2014].

[72] N. Hardy. The Confused Deputy (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22(4):36–38, Oct. 1988.

[73] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security*, 2012.

[74] HiAPK. Android Market. http://www.hiapk.com. [Online; accessed March 14, 2014].

[75] C. Jarabek, D. Barrera, and J. Aycock. ThinAV: Truly Lightweight Mobile Cloud-based Anti-malware. In *ACSAC*, 2012.

[76] A. Jø sang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, Mar. 2007.

[77] Jon Oberheide and Charlie Miller. Dissecting the Android Bouncer. https://jon.oberheide.org/files/summercon12-bouncer.pdf, 2012. [Online; accessed January 27, 2014].

[78] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in android. In *ACM SPSM*, 2012.

[79] C. M. Karat, C. Brodie, and J. Karat. Usability Design and Evaluation for Privacy and Security Solutions. In L. Cranor and S. Garfinkel, editors, *Security and Usability*. O'Reilly, 2005.

[80] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM CCS*, 1994.

[81] B. Laurie, A. Langley, and E. Kasper. RFC 6962 (Experimental) - Certificate Transparency. http://www.ietf.org/rfc/rfc6962.txt, June 2013. [Online; accessed April 14, 2014].

[82] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers. In *NDSS*, 2013.

[83] Lookout Security. Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. https://blog.lookout.com/blog/2010/12/29/geinimi_trojan, December 2010. [Online; accessed March 4, 2014].

[84] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX ATC*, 2001.

[85] S. Lucks, E. Zenner, A. Weimerskirch, and D. Westhoff. Concrete security for entity recognition: The Jane Doe protocol. In *INDOCRYPT*, 2008.

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

[87] A. Menezes, P. C. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC, 1997.

[88] Mikandi. Adult App Store. http://www.mikandi.com. [Online; accessed March 14, 2014].

[89] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *USENIX Security*, 2011.

[90] Moxie Marlinspike. Convergence (Beta). http://convergence.io/index.html. [Online; accessed March 13, 2014].

[91] Mozilla. Bug 562843 - Android Release Signing (Buzgilla@Mozilla). https://bugzilla.mozilla.org/show_bug.cgi?id=562843. [Online; accessed April 15, 2014].

[92] Mozilla. Add-on Policies: Review Process. https://addons.mozilla.org/en-US/developers/docs/policies/reviews, 2014. [Online; accessed January 24, 2014].

[93] National Vulnerability Database. Skype for Android stores sensitive user data without encryption in SQLite3 databases that have weak permissions. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1717, April 2011. [Online; accessed February 10, 2014].

[94] S. Oaks. *Java Security*. O'Reilly Media, second edition, 2001.

[95] J. Oberheide, E. Cooke, and F. Jahanian. Rethinking antivirus: Executable analysis in the network cloud. In *USENIX HotSec*, 2007.

[96] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *USENIX Security*, 2008.

[97] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC*, 2009.

[98] Osamu Aoki. Debian Reference - Chapter 2. Debian Package Management. http://www.debian.org/doc/manuals/debian-reference/ch02.en.html, 2014. [Online; accessed January 23, 2014].

[99] A. Perrig and D. Song. Hash visualization: A new technique to improve real-world security. In *Workshop on Cryptographic Techniques and E-Commerce*, 1999.

[100] Rich Cannings. Exercising Our Remote Application Removal Feature. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html, June 2010. [Online; accessed February 28, 2014].

[101] A. D. Rubin. Trusted distribution of software over the Internet. In *NDSS*, 1995.

[102] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark. FireDroid: Hardening Security in Almost-Stock Android. In *ACSAC*, 2013.

[103] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Communications of the ACM*, 17(7), 1974.

[104] Samsung. Device protection with KNOX. https://www.samsungknox.com/en/solutions/knox/technical, 2014. [Online; accessed March 19, 2014].

[105] J. Samuel and J. Cappos. Package managers still vulnerable: how to protect your systems. *USENIX ;login:*, 34(1):7–15, 2009.

[106] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *ACM CCS*, 2010.

[107] R. S. Sandhu and P. Samarati. Access Control: Principles and Practice. *Communications Magazine, IEEE*, (September):40–48, 1994.

[108] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *NDSS*, 2011.

[109] V. Shoup. Practical Threshold Signatures. In *EUROCRYPT*, 2000.

[110] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, 2013.

[111] Symantec. Android Code Signing - Digital signatures for Android. http://www.symantec.com/en/ca/verisign/code-signing/android. [Online; accessed April 15, 2014].

[112] Symbian. SymbianSigned Test Criteria. http://web.archive.org/web/20130423194443/http://www.developer.nokia.com/Community/Wiki/Symbian_Signed_Test_Criteria, 2013. [Online; accessed January 24, 2014].

[113] The Apache Software Foundation. Apache Santuario. http://santuario.apache.org. [Online; accessed April 15, 2014].

[114] The Wall Street Journal. iPhone Software Sales Take Off: Apple's Jobs. http://online.wsj.com/news/articles/SB121842341491928977, 2008. [Online; accessed March 22, 2014].

[115] Thomas Ricker. iPhone Hackers: "we have owned the filesystem". http://www.engadget.com/2007/07/10/iphone-hackers-we-have-owned-the-filesystem/, July 2007. [Online; accessed April 14, 2014].

[116] H. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, and S. Bhattacharya. The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators. In *WWW*, 2014.

[117] P. C. van Oorschot and G. Wurster. Reducing Unauthorized Modification of Digital Objects. *IEEE Transactions on Software Engineering*, 38(1):191–204, Jan. 2012.

[118] VirusShare.com. Because Sharing is Caring. http://virusshare.com. [Online; accessed March 14, 2014].

[119] VirusTotal. Free Online Virus, Malware and URL Scanner. https://www.virustotal.com, 2014. [Online; accessed April 14, 2014].

[120] W3C. W3C Standard: XML signature best practices. http://www.w3.org/TR/xmldsig-bestpractices, 2008. [Online; accessed April 14, 2014].

[121] W3C. W3C Standard: XML signature syntax and processing. http://www.w3.org/TR/xmldsig-core/, 2008. [Online; accessed April 14, 2014].

[122] D. Wallach and E. Felten. Understanding Java Stack Inspection. In *IEEE Security & Privacy*, 1998.

[123] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *USENIX ATC*, 2008.

[124] C. Wharton, J. Rieman, C. Lewis, and P. Polson. The cognitive walkthrough method: A practitioner's guide. In *Usability Inspection Methods*. Wiley & Sons, 1994.

[125] A. Whitten and J. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In *USENIX Security*, 1999.

[126] G. Wurster. *Security Mechanisms and Policy for Mandatory Access Control in Computer Systems*. Phd thesis, Carleton University, 2010.

[127] G. Wurster and P. C. van Oorschot. Self-Signed Executables: Restricting Replacement of Program Binaries by Malware. In *USENIX HotSec*, 2007.

[128] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In *IEEE Security & Privacy*, 2014.

[129] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security*, 2012.

[130] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, Scalable Detection of Piggybacked Mobile Applications. In *ACM CODASPY*, 2013.

[131] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *ACM CODASPY*, 2012.

[132] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Security & Privacy*, May 2012.

[133] 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.

[134] P. Zimmermann and J. Callas. The Evolution of PGP's Web of Trust. In *Beautiful Security*, chapter 7, pages 107–130. O'Reilly Media, 1st edition, 2009.

Appendix A

Cognitive Walkthrough of the Google Authenticator Upgrade Process

A.1 Walkthrough Setup

To illustrate the deficiencies in the process currently required to modify an application's signing certificate(s), such as, by requiring users to be involved in updating the application, we perform a cognitive walkthrough [124] of the update process as implemented by Google when switching signing keys for their Authenticator app (see Section 6.3). We refer to the versions of Authenticator with the initial signing key certificate[1] as Auth1 and the versions with the current (as of April 2014) certificate[2] as Auth2. Technically, these are considered by the OS to be distinct applications and thus must each have a unique package name[3]. However, users never see package names on Android. Apps on the Google Play store are displayed to the user with an app name and icon, which are identical in both Auth1 and Auth2.

A cognitive walkthrough aims to shed light on the user experience of performing a

[1] https://androidobservatory.org/cert/38918A453D07199354F8B19AF05EC6562CED5788

[2] https://androidobservatory.org/cert/24BB24C05E47E0AEFA68A58A766179D9B613A600

[3] Respectively: `com.google.android.apps.authenticator` and `com.google.android.apps.authenticator2`

specific task by relying on the interface for guidance. In a cognitive walkthrough, the evaluator (both a domain and usability expert) performs the core tasks required of the user and evaluates the experience against a set of guidelines or heuristics.

We consider a single core task: migrating from an installation of Auth1 to a fully functional installation of only Auth2. Since the core task is software installation, we looked to the literature for usability guidelines for installation, rather than regular software use, and borrow the installation-relevant guidelines from a cognitive walkthrough of the installation and use of Tor [34]. These guidelines, in turn curated from the literature, are:

(G1) Users should be aware of the steps they have to perform to complete a core task [125].

(G2) Users should be able to determine how to perform these steps [124, 125].

(G3) Users should know they have successfully completed each core task [124, 79].

(G4) Users should be able to recognize, diagnose, and recover from non-critical errors [124].

(G5) Users should not make dangerous errors from which they cannot recover [125].

(G6) Users should be comfortable with the terminology used in any interface dialogues or documentation [124, 79].

A.2 Evaluation

Since Auth2 is technically a new installation instead of an update (we assume users can perform standard updates), it will not appear as an update in the Play Store.

Thus, a user of Auth1 must first become aware of the existence of Auth2 through some other means (G1). Upon launching the latest (and last) version of Auth1 (v0.91), the user will encounter a prominent ribbon bar displayed at the top of the screen noting that the app will "no longer be supported." The phrase "Learn More" is offered as a link. The warning conforms to G6 but does not communicate the idea that a new version is available (G1), as opposed to the app simply being abandoned. Users may grasp that no more updates will be issued and then henceforth ignore the ribbon, never completing the core task of updating to Auth2.

If the user taps on "Learn More", they are then informed in plain language (G6) that a new version is available and are directed to the Play Store to download it. This information is sufficient for G1 and G2, and should be displayed directly in the app screen without requiring a user click-through to read it. The Play Store page for the application displays no information that distinguishes Auth2 from the already installed Auth1—it has the same app name and icon, and no language about the unusual update process for this particular app is present. A user may conclude they already have the app (contra G3). Diligent users, however, will notice the install button, which does not appear if an app is already installed (it is replaced with the option to open or uninstall).

If the user clicks to install Auth2, the app installs, automatically launches and transfers the user data from Auth1 to Auth2. (Technically, arranging for the app to open without a user click and securely transfer the data, which is private data used for authentication, requires sophisticated instrumentation of both apps by very good developers.) This automation prevents dangerous errors (G5). The user is then notified in plain language (G6) that the data has been transferred (G3) and is prompted to "uninstall the prior version of the app" (G1 and G2).

If the user clicks to uninstall the app, the OS displays a dialogue containing the app icon, app name (Authenticator), and question "do you want to uninstall this app" (G6)? In isolation, this screen does not adequately communicate to the user that the prior version is being uninstalled. Were the user instead to cancel the prompt to uninstall, perhaps believing that they do not want to uninstall what appears to be the exact app they just installed (based on the name and icon), they would have on their homescreen two identical icons with identical names and no indication of which is Auth1 and which is Auth2. If they manually uninstalled Auth2, they will

lose their data (G5). However, if they opened Auth1, a warning would appear stating that the new version is already installed and offering to uninstall this version (G4). In addition, the user data is no longer available and the app is no longer functional (G4).

The user may successfully complete the core task by uninstalling Auth1 by following the instructions to do so when prompted during the installation of Auth2, or at any time later by following the prompts in either Auth1 or Auth2.

A.3 Interpretation of Results

The intention of this cognitive walkthrough is not to criticize Google's handling of Authenticator's certificate migration. If anything, the process was relatively seamless, much of it automated, with care given to preventing dangerous errors and allowing recoverability. While there is room for improvement, this represents a nearly ideal execution of the certificate update process under the constraints of the existing OS. However in the hands of less skilled developers (*e.g.,* without clear instructions or the automation of the data transfer process upon install), the process could be much more difficult for users. Since Android currently leaves this migration process to app developers, we are apprehensive of how much worse (with regards to security and usability) a less thoughtful execution could be, and note that a consequence of user error could be data loss.

By contrast, Baton (see Chapter 6) removes all the uncertainty of developer execution and user behaviour from the equation. With Baton, the same core task can be accomplished through a standard update indistinguishable from any other update, which we assume a user can perform. Thus we can conclude, even without a cognitive walkthrough or user study, that any user able to update apps can use Baton to complete the core task.

Appendix B

List of Android Permissions

As of December 2013, applications written for Android may request additional items of functionality by requesting permissions. As of writing, the most recent API level (19) lists 145 permissions [69]. The list, available on the Android developers website, is provided below for quick reference since permissions are frequently listed when discussing Android throughout this thesis. We note that all permissions can be requested by applications, but the system may not grant the permission in all cases (even in the case of user approval). Some permissions are only given to system applications (*i.e.,* applications that are installed on the system partition). Finally we remind the reader that new permissions may be defined by developers themselves, so the total number of permissions has no upper bound.

Permission	Description
ACCESS CHECKIN PROPERTIES	Allows read/write access to the "properties" table in the checkin database, to change values that get uploaded.
ACCESS COARSE LOCATION	Allows an app to access approximate location derived from network location sources such as cell towers and Wi-Fi.
ACCESS FINE LOCATION	Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.

ACCESS LOCATION EXTRA COMMANDS	Allows an application to access extra location provider commands.
ACCESS MOCK LOCATION	Allows an application to create mock location providers for testing.
ACCESS NETWORK STATE	Allows applications to access information about networks.
ACCESS SURFACE FLINGER	Allows an application to use SurfaceFlinger's low level features.
ACCESS WIFI STATE	Allows applications to access information about Wi-Fi networks.
ACCOUNT MANAGER	Allows applications to call into AccountAuthenticators.
ADD VOICEMAIL	Allows an application to add voicemails into the system.
AUTHENTICATE ACCOUNTS	Allows an application to act as an AccountAuthenticator for the AccountManager.
BATTERY STATS	Allows an application to collect battery statistics.
BIND ACCESSIBILITY SERVICE	Must be required by an AccessibilityService, to ensure that only the system can bind to it.
BIND APPWIDGET	Allows an application to tell the AppWidget service which application can access AppWidget's data.
BIND DEVICE ADMIN	Must be required by device administration receiver, to ensure that only the system can interact with it.
BIND INPUT METHOD	Must be required by an InputMethodService, to ensure that only the system can bind to it.
BIND NFC SERVICE	Must be required by a HostApduService or OffHostApduService to ensure that only the system can bind to it.

BIND NOTIFICATION LISTENER SERVICE	Must be required by an NotificationListenerService, to ensure that only the system can bind to it.
BIND PRINT SERVICE	Must be required by a PrintService, to ensure that only the system can bind to it.
BIND REMOTEVIEWS	Must be required by a RemoteViewsService, to ensure that only the system can bind to it.
BIND TEXT SERVICE	Must be required by a TextService.
BIND VPN SERVICE	Must be required by a VpnService, to ensure that only the system can bind to it.
BIND WALLPAPER	Must be required by a WallpaperService, to ensure that only the system can bind to it.
BLUETOOTH	Allows applications to connect to paired bluetooth devices.
BLUETOOTH ADMIN	Allows applications to discover and pair bluetooth devices.
BLUETOOTH PRIVILEGED	Allows applications to pair bluetooth devices without user interaction.
BRICK	Required to be able to disable the device (very dangerous!).
BROADCAST PACKAGE REMOVED	Allows an application to broadcast a notification that an application package has been removed.
BROADCAST SMS	Allows an application to broadcast an SMS receipt notification.
BROADCAST STICKY	Allows an application to broadcast sticky intents.
BROADCAST WAP PUSH	Allows an application to broadcast a WAP PUSH receipt notification.

CALL PHONE	Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call being placed.
CALL PRIVILEGED	Allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed.
CAMERA	Required to be able to access the camera device.
CAPTURE AUDIO OUTPUT	Allows an application to capture audio output.
CAPTURE SECURE VIDEO OUTPUT	Allows an application to capture secure video output.
CAPTURE VIDEO OUTPUT	Allows an application to capture video output.
CHANGE COMPONENT ENABLED STATE	Allows an application to change whether an application component (other than its own) is enabled or not.
CHANGE CONFIGURATION	Allows an application to modify the current configuration, such as locale.
CHANGE NETWORK STATE	Allows applications to change network connectivity state.
CHANGE WIFI MULTICAST STATE	Allows applications to enter Wi-Fi Multicast mode.
CHANGE WIFI STATE	Allows applications to change Wi-Fi connectivity state.
CLEAR APP CACHE	Allows an application to clear the caches of all installed applications on the device.
CLEAR APP USER DATA	Allows an application to clear user data.
CONTROL LOCATION UPDATES	Allows enabling/disabling location update notifications from the radio.
DELETE CACHE FILES	Allows an application to delete cache files.

DELETE PACKAGES	Allows an application to delete packages.
DEVICE POWER	Allows low-level access to power management.
DIAGNOSTIC	Allows applications to RW to diagnostic resources.
DISABLE KEYGUARD	Allows applications to disable the keyguard.
DUMP	Allows an application to retrieve state dump information from system services.
EXPAND STATUS BAR	Allows an application to expand or collapse the status bar.
FACTORY TEST	Run as a manufacturer test application, running as the root user.
FLASHLIGHT	Allows access to the flashlight.
FORCE BACK	Allows an application to force a BACK operation on whatever is the top activity.
GET ACCOUNTS	Allows access to the list of accounts in the Accounts Service.
GET PACKAGE SIZE	Allows an application to find out the space used by any package.
GET TASKS	Allows an application to get information about the currently or recently running tasks.
GET TOP ACTIVITY INFO	Allows an application to retrieve private information about the current top activity, such as any assist context it can provide.
GLOBAL SEARCH	This permission can be used on content providers to allow the global search system to access their data.
HARDWARE TEST	Allows access to hardware peripherals.
INJECT EVENTS	Allows an application to inject user events (keys, touch, trackball) into the event stream and deliver them to ANY window.

INSTALL LOCATION PROVIDER	Allows an application to install a location provider into the Location Manager.
INSTALL PACKAGES	Allows an application to install packages.
INSTALL SHORTCUT	Allows an application to install a shortcut in Launcher.
INTERNAL SYSTEM WINDOW	Allows an application to open windows that are for use by parts of the system user interface.
INTERNET	Allows applications to open network sockets.
KILL BACKGROUND PROCESSES	Allows an application to call killBackgroundProcesses(String).
LOCATION HARDWARE	Allows an application to use location features in hardware, such as the geofencing api.
MANAGE ACCOUNTS	Allows an application to manage the list of accounts in the AccountManager.
MANAGE APP TOKENS	Allows an application to manage (create, destroy, Z-order) application tokens in the window manager.
MANAGE DOCUMENTS	Allows an application to manage access to documents, usually as part of a document picker.
MASTER CLEAR	Not for use by third-party applications.
MEDIA CONTENT CONTROL	Allows an application to know what content is playing and control its playback.
MODIFY AUDIO SETTINGS	Allows an application to modify global audio settings.
MODIFY PHONE STATE	Allows modification of the telephony state & power on, mmi, etc.
MOUNT FORMAT FILESYSTEMS	Allows formatting file systems for removable storage.
MOUNT UNMOUNT FILESYSTEMS	Allows mounting and unmounting file systems for removable storage.

NFC	Allows applications to perform I/O operations over NFC.
PERSISTENT ACTIVITY	This constant was deprecated in API level 9. This functionality will be removed in the future; please do not use. Allow an application to make its activities persistent.
PROCESS OUTGOING CALLS	Allows an application to monitor, modify, or abort outgoing calls.
READ CALENDAR	Allows an application to read the user's calendar data.
READ CALL LOG	Allows an application to read the user's call log.
READ CONTACTS	Allows an application to read the user's contacts data.
READ EXTERNAL STORAGE	Allows an application to read from external storage.
READ FRAME BUFFER	Allows an application to take screen shots and more generally get access to the frame buffer data.
READ HISTORY BOOKMARKS	Allows an application to read (but not write) the user's browsing history and bookmarks.
READ INPUT STATE	This constant was deprecated in API level 16.The API that used this permission has been removed.
READ LOGS	Allows an application to read the low-level system log files.
READ PHONE STATE	Allows read only access to phone state.
READ PROFILE	Allows an application to read the user's personal profile data.
READ SMS	Allows an application to read SMS messages.
READ SOCIAL STREAM	Allows an application to read from the user's social stream.
READ SYNC SETTINGS	Allows applications to read the sync settings.

READ SYNC STATS	Allows applications to read the sync stats.
READ USER DICTIONARY	Allows an application to read the user dictionary.
REBOOT	Required to be able to reboot the device.
RECEIVE BOOT COMPLETED	Allows an application to receive the ACTION BOOT COMPLETED that is broadcast after the system finishes booting.
RECEIVE MMS	Allows an application to monitor incoming MMS messages, to record or perform processing on them.
RECEIVE SMS	Allows an application to monitor incoming SMS messages, to record or perform processing on them.
RECEIVE WAP PUSH	Allows an application to monitor incoming WAP push messages.
RECORD AUDIO	Allows an application to record audio.
REORDER TASKS	Allows an application to change the Z-order of tasks.
RESTART PACKAGES	This constant was deprecated in API level 8. The restartPackage(String) API is no longer supported.
SEND RESPOND VIA MESSAGE	Allows an application (Phone) to send a request to other applications to handle the respond-via-message action during incoming calls.
SEND SMS	Allows an application to send SMS messages.
SET ACTIVITY WATCHER	Allows an application to watch and control how activities are started globally in the system.
SET ALARM	Allows an application to broadcast an Intent to set an alarm for the user.

SET ALWAYS FINISH	Allows an application to control whether activities are immediately finished when put in the background.
SET ANIMATION SCALE	Modify the global animation scaling factor.
SET DEBUG APP	Configure an application for debugging.
SET ORIENTATION	Allows low-level access to setting the orientation (actually rotation) of the screen.
SET POINTER SPEED	Allows low-level access to setting the pointer speed.
SET PREFERRED APPLICATIONS	This constant was deprecated in API level 7. No longer useful, see addPackageToPreferred(String) for details.
SET PROCESS LIMIT	Allows an application to set the maximum number of (not needed) application processes that can be running.
SET TIME	Allows applications to set the system time.
SET TIME ZONE	Allows applications to set the system time zone.
SET WALLPAPER	Allows applications to set the wallpaper.
SET WALLPAPER HINTS	Allows applications to set the wallpaper hints.
SIGNAL PERSISTENT PROCESSES	Allow an application to request that a signal be sent to all persistent processes.
STATUS BAR	Allows an application to open, close, or disable the status bar and its icons.
SUBSCRIBED FEEDS READ	Allows an application to allow access the subscribed feeds ContentProvider.
SUBSCRIBED FEEDS WRITE	No description given.
SYSTEM ALERT WINDOW	Allows an application to open windows using the type TYPE SYSTEM ALERT, shown on top of all other applications.
TRANSMIT IR	Allows using the device's IR transmitter, if available.

UNINSTALL SHORTCUT	Allows an application to uninstall a shortcut in Launcher.
UPDATE DEVICE STATS	Allows an application to update device statistics.
USE CREDENTIALS	Allows an application to request authtokens from the AccountManager.
USE SIP	Allows an application to use SIP service.
VIBRATE	Allows access to the vibrator.
WAKE LOCK	Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming.
WRITE APN SETTINGS	Allows applications to write the apn settings.
WRITE CALENDAR	Allows an application to write (but not read) the user's calendar data.
WRITE CALL LOG	Allows an application to write (but not read) the user's contacts data.
WRITE CONTACTS	Allows an application to write (but not read) the user's contacts data.
WRITE EXTERNAL STORAGE	Allows an application to write to external storage.
WRITE GSERVICES	Allows an application to modify the Google service map.
WRITE HISTORY BOOKMARKS	Allows an application to write (but not read) the user's browsing history and bookmarks.
WRITE PROFILE	Allows an application to write (but not read) the user's personal profile data.
WRITE SECURE SETTINGS	Allows an application to read or write the secure system settings.
WRITE SETTINGS	Allows an application to read or write the system settings.
WRITE SMS	Allows an application to write SMS messages.

WRITE SOCIAL STREAM	Allows an application to write (but not read) the user's social stream data.
WRITE SYNC SETTINGS	Allows applications to write the sync settings.
WRITE USER DICTIONARY	Allows an application to write to the user dictionary.