Application-centric security policies on unmodified Android

Nikhilesh Reddy[*] Jinseong Jeon[†] Jeffrey A. Vaughan[*]
Todd Millstein[*] Jeffrey S. Foster[†]

[*]*University of California, Los Angeles*
[†]*University of Maryland, College Park*

Technical Report #110017
UCLA Computer Science Department
July 5, 2011

Abstract

Google's Android platform uses a fairly standard *resource-centric* permission model to protect resources such as the camera, GPS, and Internet connection. We claim that a much better permission model for developers and users would be *application-centric*, with a vocabulary that directly relates to application-level functionality, e.g., one permission could allow camera use, but only for barcode scanning; another could allow Internet access, but only to certain domains. Despite the large apparent gap between resource- and application-centric permissions, we argue that Android already provides the necessary mechanisms to support an expressive and practical form of application-centric policies. Specifically, each application-centric permission can be represented by a new Android permission and can be enforced by coupling the permission with a trusted service running in its own process. We present a survey of the top 24 free Android apps and show that a small vocabulary of application-centric permissions covers much of the functionality of those apps. We also describe a prototype implementation of our approach.

1 Overview

Google's Android is one of the most popular smartphone platforms, with more than 100 million activated devices, more than 200,000 applications in the Android Market, and an estimated 4.5 billion apps installed from the Market [8]. Security of Android applications (henceforth "apps") is a pressing concern, as apps can collect sensitive data from the user (e.g., usernames and passwords), access personal data stored on the device (e.g., calendar and contact information), and use sensitive device capabilities (e.g., telephony, GPS, and camera).

Android takes an "open-publish" approach to app distribution, in which any app can be installed on any phone. To help address security concerns, the Android platform protects access to sensitive resources—including the camera, network sockets, and GPS receiver—with *permissions*. Each app includes an XML *manifest* file that lists the permissions requested by the app. When an app is installed, those permissions are shown to the user, who then decides whether or not to proceed with the installation. No additional permissions may be acquired when an app runs, and a security exception is raised if an app tries to access a resource without permission.

Android permissions today. While permissions on Android provide an important level of security, we have observed that, in practice, the design of Android's permission system routinely forces apps to acquire more powerful permissions than should be necessary. For example, an app that scans a product's barcode and then searches for it in a public database must have (at least) the Take Pictures and Full Internet Access permissions. As a result, apps with this feature (of which there are several in the Android Market) could potentially do much more than just barcode scanning. For example, they could access the geotag on a barcode image to find the user's location. More maliciously, they may be able to covertly capture images of a user's surroundings and transmit them anywhere on the Internet.

In our view, the basic problem with Android's permission system is that it is *resource-centric*: each permission typically controls access to a particular hardware or software resource. Thus, enforceable security policies only say *what* resources are accessed, with little or no indication of

how or *why* they are used. This leaves developers on their own to ensure they use the resources safely and only to the extent necessary. Worse, when users are presented with a list of permissions an app requests, they are left to guess at whether the app uses those permissions safely.

Application-centric permissions on unmodified Android. There are two major challenges that any solution to this issue must address: First, Android is evolving rapidly, with new hardware and software capabilities emerging regularly, and thus any solution must be agile and adaptable. Second, the permissions required by apps must capture *application-centric* security properties that are intuitively understandable to both developers and users.

It is tempting to try to address this problem by enriching Android's permission system in various ways. For example, each existing permission could be sliced into smaller permissions granting rights to correspondingly finer units of resource access. As another example, an application's manifest could use an authorization language (e.g., DCC or KeyNote) to establish constraints on resource access. A program analysis or type system (e.g., JIF) could also be used to track how information flows through an app. However, we believe such approaches require making important architectural commitments up-front, and they may be difficult to evolve on such a rapidly changing platform. Furthermore, it is imperative that the policy language be kept simple for developers and users alike.

Perhaps surprisingly, we believe that Android already contains the key ingredients needed for a powerful and practical solution to the above challenges: *interprocess communication*, *process isolation*, and *user-defined permissions*. Interprocess communication enables an application to access rich functionality provided by trusted third parties. Process isolation ensures that applications only access that functionality through a well-defined interface, thereby allowing third parties to enforce arbitrarily expressive application-centric security policies. Finally, user-defined permissions allow these policies to be associated with simple Android permissions that applications must acquire to access the desired functionality.

Consider again the problem of supporting safe barcode scanning. An ideal security policy would specify that the camera may only be used to scan a barcode, and the resulting images are thrown away after processing. We propose to represent this policy as a new Android permission, ScanBarcodes, that grants access to a trusted library that obeys the policy. To do so, the library could have a single function that displays the current camera image, waits for a user click, and then scans the resulting image for a barcode, and returns the barcode's numerical value to the calling app. Furthermore, we can implement the library as an Android *service* that runs in a separate process. Therefore, while the library must be granted full camera access, an app that calls into the library need only be granted ScanBarcodes access, thereby providing a strong and understandable guarantee to both the app developer and users.

Although at first glance it seems we may need many such application-centric permissions, our hypothesis is that in practice a reasonably small set can dramatically improve the security of a wide variety of applications. Moreover, we envision an ecosystem in which many different vendors provide services associated with commonly desired application-centric permissions. These services will be far simpler than full apps and hence should be easy to audit for security, and they are attractive components for open-sourcing since they likely will not contain proprietary features of an app. Finally, by modularizing each application-centric permission in its own abstraction boundary,

we decrease the potential for harm due to policy violations in these libraries. For example, our price-checking application would naturally use one service to provide the barcode scanning and a separate service to provide access to a barcode database on the Internet, thereby greatly reducing the potential for vulnerabilities caused by the interaction of camera and Internet permissions.

To explore these ideas, we have undertaken several preliminary tasks. We performed a survey of popular Android apps to identify their application-centric policies (Section 2). We implemented our proposed approach as an Android library ACPlib, which comprises three application-centric permissions and the associated services. Finally, we developed Redexer, a Dalvik bytecode rewriting framework that retrofits downloaded apps to use application-centric policies. We describe our preliminary experience using ACPlib and Redexer to enhance the security of existing and new apps (Section 3).

2 Feasibility study

We performed a preliminary study to evaluate the extent to which application-centric permissions can be shared across a variety of apps to enforce stronger security policies.

Methodology Our feasibility study considered the top 24 free apps on Google's Android Market[1] as of April 13, 2011. These apps were selected because the Market website displays them prominently to users; they are widely installed (as reported on their Market home pages); and they represent a spectrum of application domains.

The evaluation consisted of installing and running each app to understand its functionality, reading English-language privacy policies or other documentation when available, and, sometimes, crude analysis of binaries (using the Unix `strings` command). In the case of WhatsApp, only limited functionality was tested due to restrictions on app registration. For each app, we evaluated how it uses its current permission set and identified application-centric permissions that could replace some of these permissions.

Results The results of our study are summarized in Figure 1. The top left-hand column of the table shows a selection of Android permissions requested by apps, and the bottom left-hand column shows application-centric permissions we identified as potential replacements. These permissions are described beneath the table and range from capturing specific Internet uses to restricting use of location data. We discuss several of our application-centric permissions in detail.

Internet permissions. Four of the 11 permissions pertain to the Internet. The permission InternetURL(*domain*) allows network connections only to *domain* and its subdomains. This is useful for the common case in which an app communicates with only a handful of known web services, e.g., Google's Sky Map can use InternetURL(google.com) in lieu of arbitrary Internet access.

[1] `https://market.android.com/`. The apps surveyed are Alchemy 1.10.2, Google Maps 5.4.0, Dropbox 1.1.1, GasBuddy - Find Cheap Gas 1.14, Street View on Google Maps 1.6.0.6, Angry Birds 1.5.3, Bubble Blast ! 1.0.16, Shazam 2.5.3-BB70302, ASTRO File Manager 2.5.2, Pandora Radio 1.5.5, Advanced Task Killer 1.9.6B76, Barcode Scanner 3.53, Vaulty Free Hides Pictures 2.4.1, Facebook for Android 1.5.4, FreeMusicDownloader 1.8.3 Live Holdem Poker Pro 3.01, Angry Birds Rio 1.0.0, Horoscope 1.5.2, KakaoTalk 2.0.1, Flash Player 10.2.156.12, Bubble Blast 2 ver. 1.0.18, Google Sky Map 1.6.1, and WhatsApp Messenger 2.6.2642.

	Alchemy	Angry Birds	A. Birds Rio	ASTRO	Barcode	Bubble Blast	Bub. Blast 2	Dropbox	Facebook	Flashplayer	FreeMusic	GasBuddy	Horoscope	KakaoTalk	Live Holdem	Maps	Pandora	Shazam	Sky Map	Street View	Task Killer	Vaulty	WhatsApp	YouTube
Full Internet Access	✘	✘	✘	✘	✘	✘	✘	✘	✘		•	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
Storage contents				•	•			•			✘		✘	✘	✘	✘						✘	•	
Location fine/coarse									✘		✘	✘	✘			✘			✘	✘			•	
Modify global settings				•	•						✘		✘			•	•		✘				•	
Read phone state/id	✘					✘	✘	✘			✘	•	✘	•	✘	•	•	•						✘
Take photos/videos					✘							•		•										
AdsPrivate	+	+	+	+		+	+										+				+	+		
AdsGeo											+		+											
AnonUsage		+	+			+	+	+						+				+						+
InternetURL(developer)	+	+	+		+			+	+			+	+	+	+	+	+	+	+	+			+	+
InternetURL(other)						+	+								+			+					+	+
LocationBlock												+												
LocationVisible									+							+		+	+	+				
MobileBilling															+									
ScanBarcodes					+																			
SDCardOwnFiles													+	+		+						+		
SDCardShared											+				+							+		
ToggleGPS											+		+						+					

AdsPrivate: May displays ads, but without sharing personal information with advertisers.

AdsGeo: May displays ads and may share your location, but no other personal information, with advertisers.

AnonUsage: May report anonymous usage information to its developers, including a random number identifying your copy of the app, but not you or your phone.

InternetURL(x): May access the internet services located at domain x.

LocationBlock: May access approximate location, accurate to 150m (about one city block).

LocationVisible: May acquire accurate location, but only when the app's interface is showing.

MobileBilling: May bill you via your carrier, after requesting permission with a prompt.

ScanBarcodes: May use the camera to read barcodes and QR codes.

ToggleGPS: May enable or disable the GPS receiver.

SDCardOwnFiles: May manage files on its own area of the SD card; cannot read, edit, or delete other files.

SDCardShared: May manage files, such as music or photos, that are shared by several apps; cannot read, edit, or delete that belong to other apps.

Figure 1: App-centric permissions for top 24 apps. Notation ✘ indicates a built-in Android permission that can be replaced by one or more application-centri permissions. + indicates application-centric policies to be added and • indicates policies that cannot obviously be removed. Some Android permissions, such as those related to account managment are outside the scope of this paper, and not shown.

The InternetURL permission is too coarse-grained to use for in-app advertising, since both the advertiser and the app developer have incentives to extensively share user data, violating reasonable privacy expectations. Yet totally forbidding communication with advertisers is also undesirable, as ad revenue encourages developers to release free apps. The permissions AdsPrivate and AdsGeo manage this tension by allowing advertising while restricting flows of private data. A similar AnonUsage permission is intended for the collection of general, anonymous analytics via services such as Flurry.[2]. (An alternative design could parametrize the permission by ad network.)

The application-centric Internet permissions above impose strong restrictions on Internet access while still allowing most desired functionality. Of the 23 apps that originally required Full internet access, 22 can be rewritten to use only application-centric Internet permissions. The remaining app, Freemusic, downloads media files from diverse domains and legitimately needs full Internet access.

Storage permissions. Android's default handling of external storage, such as SD cards, allows any app to modify data stored by any other app. This policy is overly broad for many apps, such as Freemusic, that should only access deliberately modify media libraries, and for others, such as Horoscope, that do not appear to legitimately need modify to shared files at all. Indeed, we believe the restrictive SDCardOwnFiles and SDCardShared policies can replace Android's built-in storage permission for six of the ten apps that require it.

GPS permissions. We found that four of the seven apps that request the Modify global settings permission seem to use it solely to toggle the GPS unit on or off, to save power (as distinct from the right to access GPS location data, protected by a different permission). These apps can be granted the more restrictive ToggleGPS permission instead. Permissions LocationBlock and LocationVisible restrict access to GPS location data in two different ways, and these permissions appear sufficient to replace Android's GPS permission in seven out of eight apps. As suggested by LocationBlock, we believe the distinction of GPS vs. network location is less interesting than the distinction between highest-precession-possible vs. intentionally-degraded location.

Overall, of the requested Android permissions we studied, 71% are replaceable with application-specific permissions that are much more restrictive, and yet should not adversely affect functionality. The permissions InternetURL, AdsPrivate, and AnonUsage are applicable to at least 1/3 of surveyed apps, and InternetURL itself is applicable to 2/3. Finally, 8 of the 11 permissions are applicable to at least 10% of the surveyed apps. This study therefore provides preliminary evidence that for many Android apps, a small number of application-centric permissions can provide significantly stronger security guarantees without loss of functionality.

Implementing application-centric permissions The 11 application-centric permissions we identified are intended to be enforceable by interposing a strong API, implemented via a service, between underlying resources and clients apps. To give a flavor of how that might work, we sketch how two of the permissions could be enforced by a trusted service. For purposes of exposition we elide some details, notably Android's event driven programming model and pervasive use of objects. (The prototype described in Section 3 does follow Android's programming model.)

First, consider the InternetURL(*domain*) permission, which allows an app to connect to (sub-

[2] `http://www.flurry.com`

domains of) *domain*. This functionality can be implemented by a service with the following interface:

```
Connection open(string url);
byte[] read(Connection c);
void write(Connection c, byte[] data);
void close(Connection c);
```

In Android, global state is used to track a security context, and `open`(x) checks the current context for a permission of the form InternetURL(y), where x is a subdomain of y. If such a permission exists, `open` connects to a socket and returns a valid `Connection` object. Otherwise, `open` raises a security exception. While this secure kernel provides few operations, wrappers can extend it to a richer interface.

Although Android does not directly support parameterized permissions such as InternetURL, these can be encoded using *permission trees*. A permission tree is a family of permissions whose names share a common prefix. For instance InternetURL(google.com) can be given full name acplib.perm.URL.google_com which is part of the acplib.perm.URL tree. Services must be instructed to preregister tree elements before client installation, but this does not appear to be a fundamental limitation of the platform.

Second, consider the AdsPrivate permission. A trusted library can mediate between apps and well-known ad services using an interface such as:

```
enum AdService { ADMOB, JUMPTAP, ...}
Connection open(AdService a);
byte[] newAd(Connection c);
void close(Connection c);
```

This interface allows ads to be displayed (via `newAd`) , but prevents the app from passing any information to an advertiser. The service could also mitigate covert timing channels using a combination of prefetching and delaying ad requests. One wrinkle is that online advertising requires that apps identify themselves using a unique id so that the right developer can be paid for clicks. The service can use the global context to identify calling apps, along with a well-known map from apps to ids that is consulted the first time an app requests an ad connection. Finally, while it would be appear difficult to implement and maintain a single multi-advertiser abstraction layer, companies such as AdWhirl[3] do this already, albeit without our security focus.

3 ACPlib and Redexer

To gain preliminary experience with some of the permissions discussed in Section 2, we implemented a prototype application-centric permission system for Android. Our system comprises two main components: ACPlib, which provides an implementation of application-centric permissions and their associated services, and Redexer, a Dalvik-to-Dalvik rewriting system that can modify apps, even without having their source code, to use ACPlib.

[3] `https://www.adwhirl.com`

ACPlib ACPlib is collection of Android services, each implementing one of the following permissions: InternetUrl, LocationBlock, or ScanBarcodes. The services listen for request messages from other clients apps and ensure client apps have appropriate privileges before servicing requests.

As described in Section 2, security dictates that ACPlib services run in separate processes from their clients, with communication only via Android's RPC mechanism. Using this directly is more complex than simply calling privileged system routines. To ameliorate this, ACPlib provides drop-in API replacements for system libraries that handle necessary RPC calls, ACPlib internally. For example, instead of calling `java.net.URLConnection.openConnection()` to open an Internet connection, users now call `apclib.net.URLConnection.openConnection()`. Additionally apps must *bind* to ACPlib, typically done in the app's `onCreate()` method.

Redexer ACPlib can be used as-is by security-conscious developers to reduce the privilege level of their apps. We also expect that app users will wish to retrofit existing apps to use ACPlib, e.g., to restrict the web sites apps can visit or coarsen the location information revealed to apps. To this end, we have begun development of Redexer, a Dalvik binary rewriting framework that modifies application bundles to replace Android API calls with ACPlib equivalents. Redexer also adds the Dalvik code for ACPlib's replacement APIs to the application.

One surprising challenge in developing Redexer is the rules that Android's verifier enforces before it will execute a Dalvik bytecode file. In particular, Dalvik files contain several indexed "identifier lists" of data that is shared across methods, e.g., strings, types, field and method definitions, etc. The Android verifier requires that such pools are both duplicate-free and sorted in a particular order. This causes some complications when adding the ACPlib API to the app's Dalvik file. For example, there must be only one string "`V`" representing the type void in a Dalvik file, and it is almost guaranteed this type will appear in both the app's code and in the ACPlib API code; thus upon merging, we must eliminate one copy and rewrite one or the other file accordingly.

Another challenge for Redexer is that some apps call ACPlib services from `onCreate()`, but (due to Android's event-driven semantics) the connection to ACPlib cannot be established until after `onCreate()` returns. Thus, Redexer splits `onCreate()` into two methods: It heuristically keeps all the code up to and including the `setContentView()` call (which sets up the user interface) in `onCreate()`, and then appends a call to perform the binding. We move the remainder of the code into a new `droidLibOnCreate()` method that is invoked by ACPlib after the binding completes. We expect to make this mechanism more robust in the future.

Preliminary Experience While ACPlib and Redexer are far from fully mature, we were able to modify the source of two existing apps to use ACPlib and to rewrite two apps automatically using Redexer. We also built a new app from scratch using ACPlib.

• *Google Translate*[4] is a very popular app that requests full Internet permissions, but only contacts the `googleapis.com` domain. We manually edited the source code of the app to use InternetURL(`gooleapis.com`) instead. We found the necessary changes easy to make, and after making the changes, the app continues to work correctly.

• *Maurauder's Map* is a route-planning app we wrote prior to ACPlib. We manually updated its source to use LocationBlock, allowing users to find reasonable routes without revealing their exact

[4] `http://code.google.com/p/apps-for-android/`

location. As before the changes were easy to make and the app continues to work well.

• *Slashdot RSS Reader*[5] is an app that contacts the `slashdot.org` domain to retrieve an RSS feed, articles, and comments. We used Redexer to rewrite the app to use InternetURL(`slashdot.org`). The domain was found automatically using Redexer to search for URLs in the binary.

• We implemented a *Price Checker* app from scratch that uses ScanBarcodes to scan barcodes and InternetURL(`searchupc.com`) to look up the price for the scanned item. This was easy to write using ACPlib's barcode scanning library.

4 Related Work

Others have also recognized the limitations of Android's resource-centric permission model. Barrera et al. [1] and Felt et al [7] analyze the way permissions are used in Android and Chrome OS apps. Both groups observe that only a small number of Android permissions are widely used but that some of these, in particular Internet permissions, are overly broad. Some researchers have developed tools that have found a variety of security issues in Android apps [4, 5]. While our approach cannot guarantee the absence of the security vulnerabilities found by such tools, we believe it can help make apps more secure in practice. We believe APClib is complimentary to such tools as they address different sorts of security properties. Furthermore, trusted libraries like ACPlib are prime candidates for automated validation, as reuse allows verification costs to be amortized and high security requirements can justify remaining per-app costs.

Others have also proposed enhanced permission mechanisms for Android. *MockDroid* changes Android OS so that users can "mock" a subset of an application's resource-centric permissions, causing accesses to those resources to silently fail [2]. *Apex* is similar and also lets the user enforce simple constraints such as the number of times per day a resource may be accessed [9]. *Kirin* employs a set of user-defined security rules to flag potential malware at install time [6]. These tools allow users to trade off app functionality for privacy, but they inherit the resource-centric nature of Android permissions, which can limit their effectiveness. For example, denying Internet access to Google Translate would render it useless, so a MockDroid user must allow such access, whereas our application-centric policy provides a much stronger guarantee. Moreover, our approach can be implemented purely as a library, with no modifications to the underlying Android OS.

Saint enriches permissions on Android to support a variety of installation constraints, e.g., a permission can include a whitelist of apps that may request it [10]. In our limited experience, we have not yet needed this capability. ComDroid [3] analyzes inter-application communication for potential security risks. This tool could complement our proposed approach, which relies heavily on inter-application communication with trusted third parties.

[5] `http://code.google.com/p/slashdot/`

5 Conclusion and Future Work

We introduced the idea of application-centric permissions and argued that they are an expressive and practical approach to increase the security of Android apps today. We believe the same idea can also be applied to other permission systems. In the future, we plan to develop a wider vocabulary of application-centric permissions; implement more permissions in ACPlib; and improve Redexer so that we can automatically rewrite more apps. We also hope to conduct a study to determine how developers and users would understand and use application-centric permissions.

References

[1] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *CCS*, pages 73–84, 2010.

[2] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *HotMobile*, 2011.

[3] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *MobiSys*, 2011. To appear.

[4] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

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

[6] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS*, pages 235–245, 2009.

[7] A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. In *WebApps*, 2011. To appear.

[8] Google. Android: momentum, mobile and more at Google I/O, May 2011. `http://googleblog.blogspot.com/2011/05/android-momentum-mobile-and-more-at.html`.

[9] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, pages 328–332, 2010.

[10] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *ACSAC*, pages 340–349, 2009.