Android Application Security

A Thorough Model and Two Case Studies: K9 and Talking Cat

Hannah Gommerstadt
Harvard University
hgommers@college.harvard.edu

Devon Long
Harvard University
dlong@college.harvard.edu

Abstract

As the smart phone craze spreads through the modern world, it is becoming crucial to reason about the flow of sensitive data on the mobile platform. Some work has already been done on the Android security model, including several analyses of the model and frameworks aimed at enforcing security standards. In this paper we use this work as a starting point as we synthesize a detailed model of the information flow within the Android platform and present two detailed case studies.

1. Introduction

Statistical research predicts that about 65% of the American population, or about 200 milllion people, will have a smart phone and/or tablet device by 2015. Given the fact that over 296 million smartphones were shipped worldwide last year, it is clear that the smart phone trend is on the rise [5]. Google's Android platform is currently the most popular platform and runs on 48.6 % of smart phones as of January 2012 [6]. As smart phones become more and more ubiqutuous, and the software industry transitions en masse over to the mobile arena, reasoning about information flow on mobile devices is going to become more and more crucial.

In order to be useful, smart phones must make use of a plethora of private data such as one's location, contacts, and system settings on the user's phone. This data is frequently used when the phone communicates with the outside world, usually via the internet, in the form of Android applications. Given the sheer amount of sensitive data that is transmitted with every click of a button, users need guarantees about the security of their personal information, as well as a clear understanding of exactly how their information is flowing through Android.

While some work has been done on Android security and information flow, there does not yet exist a cohesive model of all the flows of sensitive information within an Android application that takes into account the different sources of private data. In this paper we present an overview of work done on Android security, a more detailed information flow model and the results of two case studies which examine the information flow through two popular Android applications.

2. Android Overview

In order to present the Android security model, we first review the structure of the Android platform and the foundations of Android applications.

2.1 Platform Overview

The Android platform is a Linux-based open-source mobile platform used for phone developement which is maintained by Google. Its core functionality is written in Java, C, and C++. Applications for the platform are written mainly in Java, though some do employ native code alongside the Java application code. These applications are then compiled to DEX byte-code format via the Dalvik interpreter and executed on the device.

Using a Linux sandboxing model, each application runs in its own process with none having the permisssion to run as root. Though applications are composed of multiple components, which are discussed in Section 2.2, by deafult all components of the same application run in the same process. Consequently, each application has its own instance of the interpreter and they communicate through Inter-Process Communication mechanism. The specifics of inter-application communication are discussed in Section 3.2.

2.2 Application Overview

An Android application is composed of multiple components, which are distinct, potential locations for information leaks. It is important to understand the individual parts of an Android application in order to reason about information flow between applications in a detail-oriented manner. There are four types of components that comprise Android applications: activities, services, content providers and broadcast receivers.

The activity components are the main visual components of the application. In other words, these are the screens the user sees and interacts with while using the application. For instance, in an email client application, examples of activities would be screens that would allow the user to compose a message, take a photo, or perform other actions.

The service components of the application consist of the computations that run in the background. These are hidden from the user, but are crucial for the application to function. For instance, in an email client application, examples of services would be calls to the network to fetch and send messages and querices to determine the current status of the internet connection.

The content providers allow the application to page in a structured set of data for the application to use. Frequently, this is done in the form of a small-scale data base which the application then queries. For example, in order to access calendars, events, reminders, and so on, a Calendar Provider API is provided by Android platform. Developers have the option of building their own content providers to page in the specific data their application may require.

The broadcast recievers allow the application to react to system broadcasts such as the low battery signal, the status of the headphone jack (are the headphones plugged in or not) and other system information. For example, in a music player application if the action of headphones being removed causes the music to be played loudly through the internal microphone, the application needs to employ a broadcast reciever.

3. Security Model Overview

Figure 1 demonstates the general security model of Android Applications. In this coarse model, there are three categories of sensitive data: personal information, sensitive input devices, and device metadata.

The personal information refers to a variety of information stored on the phone that is sensitive to the user, such as their text messages, call log, photos, or contacts list. The sensitive input devices refer to phone hardware which could reveal private information, such as the accelerometer, GPS, audio system or microphone. Finally, the device metadata refers to data that uniquely identifies the phone such as its identification number, user system preferences, browser history, or the phone number associated with the phone.

We divide the discussion of this security model into three segments – the permission model, communications between applications, and the limitations of the model.

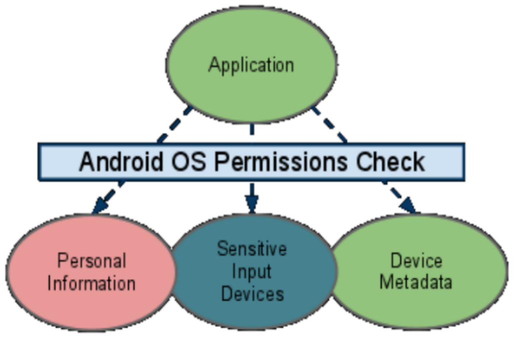


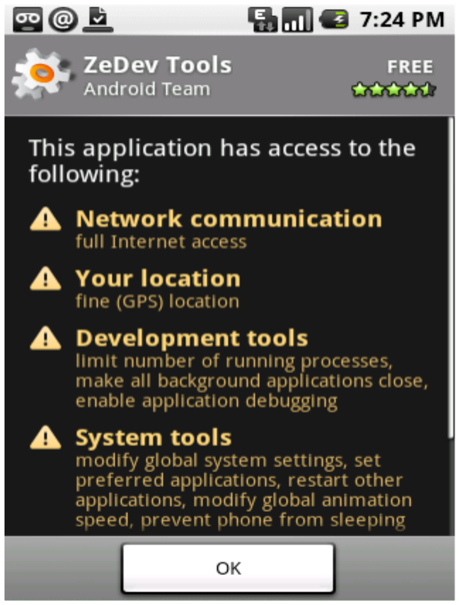
Figure 1. Android Security Overview [1]. In this diagram we see three types of sensitive information being protected from the application by means of a permissions check.

3.1 Permissions

For each application, access to potentially sensitive data is mediated by permissions granted by the user at install time. The user is prompted with a screen asking whether the application has permission to access various sources of data. An example of such a screen is shown in Figure 2. The user is then required to grant all of the permissions requested for the application to install and run successfully.

For example, an application may request access to the internet by asking for the INTERNET permission, or for the exact location of the phone by prompting the user to grant the ACCESS_FINE_LOCATION permission. The Android API provides developers with a list of over 100 default permissions to choose from. Developers are also allowed to define custom permissions for their applications which the user also has to approve at install time. These custom permissions are usually permissions to access some specific bit of data or to mediate interactions between applications.

An example of some default Android permissions and their effects can be found in Figure 3. This figure shows the default


7:24 PM
ZeDev Tools
Android Team
FREE

This application has access to the following:

Network communication
full Internet access

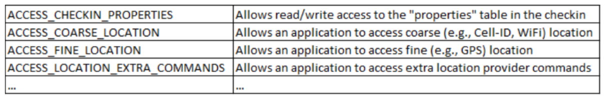
Your location
fine (GPS) location

Development tools
limit number of running processes, make all background applications close, enable application debugging

System tools
modify global system settings, set preferred applications, restart other applications, modify global animation speed, prevent phone from sleeping

Figure 2. Example of a screen prompting the user to agree to grant application permission to access sensitive data. Note that this application requests the right to access the user's location and communicate with the network. [8]
.

permissions that an application may request at install time and require the user to approve in order to run.



ACCESS_CHECKIN_PROPERTIES	Allows read/write access to the "properties" table in the checkin
ACCESS_COARSE_LOCATION	Allows an application to access coarse (e.g., Cell-ID, WiFi) location
ACCESS_FINE_LOCATION	Allows an application to access fine (e.g., GPS) location
ACCESS_LOCATION_EXTRA_COMMANDS	Allows an application to access extra location provider commands
...	...

Figure 3. Example of Permissions Defined by an Application. Note that this particular application requires access to the user's specific location.

3.2 Communication Between Applications

As mentioned previously, applications are sandboxed and run their own instances of the byte code interpreter. However, IPC is permitted and occurs over a few channels.

IPC happens by way of intent messages, objects which contain a bundle of information. This information usually involves instructions for how to process the intent – what action should be taken and on what data. This information is specific to a particular component of an application because intents are communicated to activities, services or broadcast receivers specifically, and not to the entire application.

These intents are sent from a specific component of an application (such as an activity) to another component of a different application (such as a broadcast server). For example, if the battery of the phone is low, an application may send an intent message from its broadcast receiver (which reported that the battery was running out) to the activity component of another application which would trigger a pop-up window warning the user to charge their phone.

3.3 Limitations of the Model

The main limitation of this model is that the permissions are very broad. The INTERNET permission allows the user to access any URL on the internet, not just a specific one that the application may need. Further, many applications request permission to access the phone's unique identification number, when many could do just as well with some coarser data.

A study of Android applications which reviewed 68% of all applications on the market revealed that 1 in 5 applications request personal information and that 1 in 20 applications have permission to make any phone call they would like [7]. These types of statistics highlight the fact that there are some problems with the current security model.

4. Literature Review

We separate the overview of recent work on this topic into research that aimed to provide an overview of the Android security model or expose problems in the model, and research which strived to develop better security frameworks for Android.

4.1 Android Security Analyses

A Study of Android Application Security [10] provided an excellent overview of the high level security issues present on the Android platform. The authors of this paper argued that the security characteristics of Android applications are not yet well understood and more research is necessary to determine the exact flows of information through Android devices.

In an effort to familialarize themselves with exactly how Android application security works, the authors chose 1100 very popular applications, which consisted of 21 million lines of code, and decompiled them back into Java. After decompiling them into Java, the authors performed some static analysis on the code. They concluded that while there is great evidence of vulnerability of private information, there is little evidence of vulnerabilites that lead to malicious control of the phone.

Blackhat [8] was a technical report that provided deeper insights into the Android security model. It provided developers with information on how to go about writing secure applications, and delved into the technical details of the security flaws of the current Android model. The author stated that Android's basic security, which he characterizes as just sandboxing without giving any application root privileges, is not sufficient.

The report came to the conclusion that malicious software is an unfortunate reality on popular platforms, but Android does try to minimize the impact of malware through its features. However, even unprivileged malware that gets installed on an Android device (perhaps by pretending to be a useful application) can still temporarily wreck the users experience.

4.2 Frameworks

In this section we discuss the research frameworks which have been developed to track information flow on the Android device, or which attempt to enforce stricter permissions on the device.

4.2.1 TaintDroid

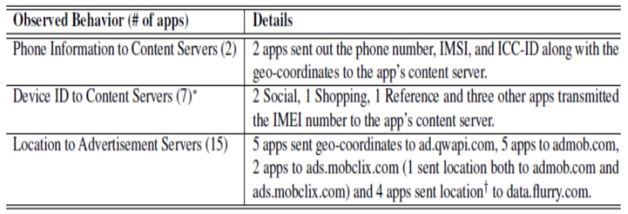
TaintDroid [9] was developed as a system to help track the flow of secure information in Android devices. The developers of TaintDroid tried to tackle the reality of third party applications becoming normative in the mobile world. These applications frequently access private user data, and in many cases these third parties should not be trusted.

At a conceptual level, TaintDroid applies labels to secure data, such as one's private contacts list or SMS messages. When this information leaves the device, it is logged by TaintDroid. This mechanism aims to provide greater insight into exactly how information flows through the Android platform as the user interacts with various applications.

TaintDroid was implemented to run in the background. It does not interfere while the user interacts with applications; it just logs the information flow. In practice, TaintDroid was shown to have a less than 14% CPU overhead.

TaintDroid was applied to 30 popular Android applications. It flagged the flows of secure information, which the authors later examined for malicious intent. The results are summarized in Figure 5. The findings can also be summarized with the statement "Two thirds of these applications expose detailed location data, the phone's unique ID, and the phone number using the combination of the seemingly innocuous permissions granted at install" [9].

The main drawback of TaintDroid is that it only tracks data flows, and not control flows, which does limit the information that it can log. This limitation is consistent with the notion that only static analysis can fully track control flow.



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

Figure 4. Security Violations Found by TaintDroid [9]. These results show that more than 20 of the analyzed applications send phone information (such as the phone number and the unique phone ID) to various servers. They also sent geographic coordinates to advertisement servers.

4.2.2 Dr. Android and Mr. Hide

Dr.Android and Mr.Hide [11] was created to restrict how applications access private information. The developers of this framework tried to address the issue of Android security permissions being overly broad. They claimed that this permission model led to unecessary security vulnerabilities. For example, the Amazon application (which allows users to interact with the Amazon.com website) requests permission to access any website on the internet – not just amazon.com, which the only website it needs to do its job.

Their solution was to replace existing overly broad permissions with finer grained permissions, and perform a static analysis of the code. Their dual pronged approach had "Mr. Hide" be the only proces that could access secure data from the phone and "Dr.Android" be a program which would transform existing applications so that they could only access sensitive data via "Mr. Hide". This approach is summarized in Figure 5. Their implementation of this model would involve installing "Mr. Hide" and having this processs always be running. Then "Dr. Android" would be run on each application that the user wants to use. "Dr.Android" takes about a minute to transform the original application to a "Mr. Hide" compatible application and there is no further overhead associated with this framework.

This framework was tested on 19 applications and identified 66 uses of dangerous permissions. 40 of these permissions were automatically removed by Mr. Hide/Dr. Android and replaced with finer grained permissions. This framework assures that an application gets no more permission than it needs to function, but makes the assumption that the application itself is trusted.

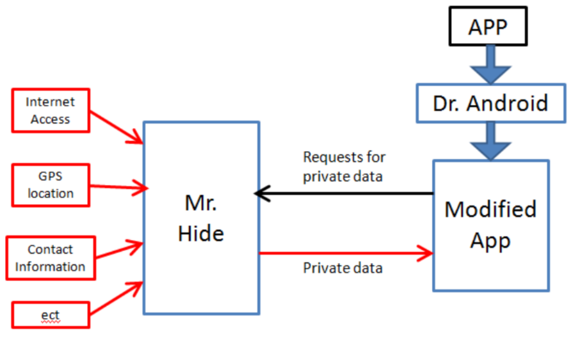


Figure 5. Dr. Android and Mr. Hide Model. This figure aims to visually portray the interactions of Dr.Android, Mr.Hide, the application and the user's sensitive data.

5. Detailed Model

In this section we present a more thorough discussion of the Android permission model. We begin with a discussion of the sources of private data on Android devices, and then consider how this private information can flow within an application, among applications, and ultimately to the outside world. Figure 6 summarizes this detailed model which illustrates all the sources of private data, how this private data can flow, and how permissions restrict these flows.

5.1 Sources of Private Data

There are three possible sources of private data in the Android framework: device hardware, default Android software, and user added Applications.

Sensitive user data originating from device hardware includes the user's GPS coordinates (whether they be coarse or fine grained), the current state of the wireless connection, information from the accelerometer, the battery status, and data caputured from the camera or microphone of the device, as well as data from many other hardware sources.

In addition to the hardware, Android devices come with several default Applications (phone book, default email client, etc) that collect user data. Data of this sort includes contacts, calendar, browser history, SMS messages, MMS messages, calls, and system messages.

Finally, user data can originate in third-party applications that users download and install on their devices. This occurs when the user provides information to the application, usually through a form of some sort. Examples include a user's shopping list in an online shopping application, or a to-do list in an organizer application.

5.2 Permission Model

In the Android security model permissions restrict the flow of information (1) from hardware to applications, (2) between applications and the network, and (3) among applications. As explained in greater detail below, if an application wants to access a piece of hardware, call a component of another application, or talk to the network, it must do so through an API call, and have the corresponding permission.

Permissions can be default Android permissions, or custom permissions created by developers. The default Android permissions are associated with the hardware (the ACCESS_FINE_LOCATION permission is associated with GPS access) or default applications and software (the READ_CONTACTS permission is associated with access to the user's contact list).

Developers can also create custom permissions using <permission> tags (in the manifest.xml file). These custom permissions are treated just like the default permissions: if some component of an application is protected by a custom permission, and another application tries to access that component it must hold that specific permission. An example of a custom permission might be a READ_SHOPPING_LIST permission that protects access to a user's shopping list in an online shopping application. A developer of an application that collects user data and can share this data with other applications through IPC should always create and employ appropriate custom permissions.

The permissions are all specified in the manifest.xml file. Permissions that an application requires are specified in <uses-permission> tags. At the time of installation, the user is presented with the list of permissions that an application is requesting. If the user feels comfortable granting the application all the permissions it requests, then the application is installed with the permissions. An application can obtain no new permissions after install time. If the user does not feel comfortable granting the permissions an application requests, the application is not installed.

In addition to specifying the permissions they need, applications must specify if any of their components are protected by permissions (default Android permissions or perhaps custom permissions). Again, this is done in the manifest.xml. A developer can protect an activity, service, or broadcast receiver with a permission by listing it in the permission field of the <activity>, <service>, or <receiver> tag.

Finally, the custom permissions an application creates must be declared in the manifest.xml file. This is done using the <permission> tag. The developer must specify a name and protection level for the permission (in the name and protectionLevel fields of the <permission> tag). The name is simply a unique identifier for the permission. The protection level is one of four values (normal, dangerous, signature, signatureOrSystem) that specify how dangerous the permission is. Figure 7 summarizes the differences between these categories.

5.2.1 Flows from Hardware

With the permission model in mind, we can now consider how permissions, along with the Android API, restrict or permit flows of information. The first type of flows we consider is flows from hardware. Again, hardware sources of private information are devices such as the GPS or the accelerometer.

Hardware can only be accessed through the Android API. For example, the GPS is accessed through the LocationManager class. In order to make calls through the API, an application needs the corresponding permission. Continuing with the GPS example, an application needs the ACCESS_FINE_LOCATION permission (which is one of the default Android permissions) in order to make calls to the LocationManager class.

This system of an API and a corresponding permission keeps information flows from hardware simple. If an application wants to access information from hardware directly, it can do so only if it has the corresponding permission.

5.2.2 Network Flows

The second type of information flow we consider is flows from applications to the network, which are very similar to the flows from hardware to applications. Network sockets can only be opened using the Android API, and access to these APIs are restriced by the INTERNET permission (a default Android permission).

Specifically, an application with the INTERNET permission can access any website (there is no sub permission that restricts

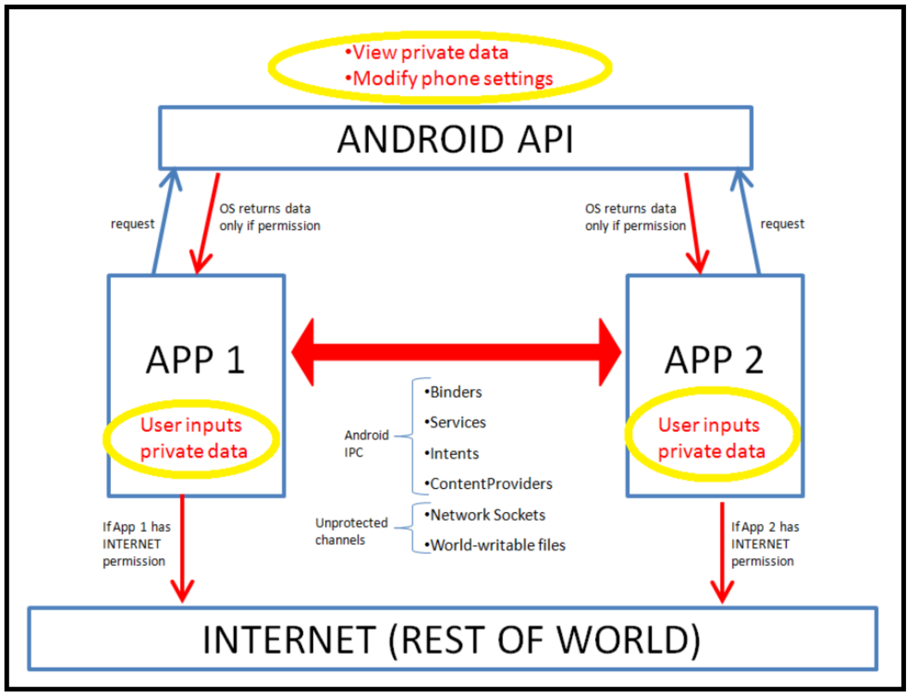
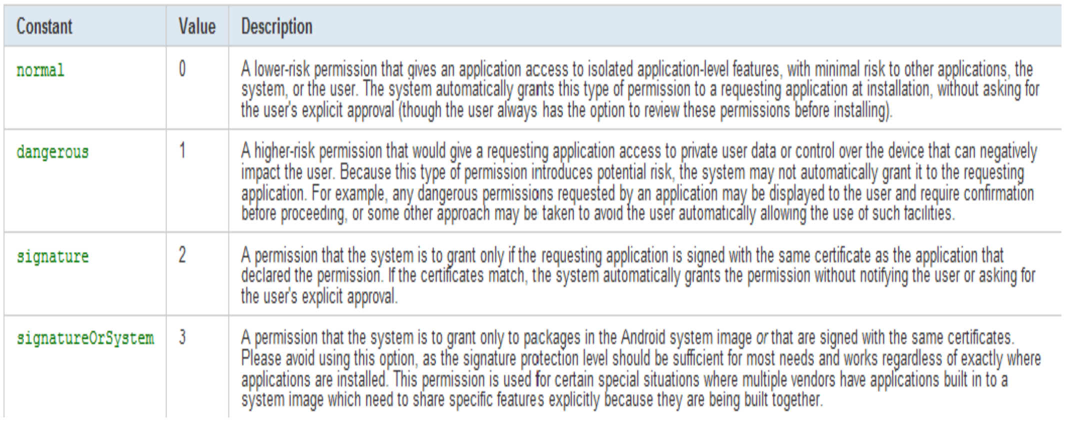


Figure 6. Detailed Security Model. Sources of private data (hardware, default software, and information inputted into 3rd-party applications) are circled in yellow. Red arrows indicate flows from sources to applications, from applications to the network, and between applications (both IPC and non-IPC). The text summarizes how the Android API and permissions restrict or permit these flows.



Constant	Value	Description
`normal`	0	A lower-risk permission that gives an application access to isolated application-level features, with minimal risk to other applications, the system, or the user. The system automatically grants this type of permission to a requesting application at installation, without asking for the user's explicit approval (though the user always has the option to review these permissions before installing).
`dangerous`	1	A higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. Because this type of permission introduces potential risk, the system may not automatically grant it to the requesting application. For example, any dangerous permissions requested by an application may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities.
`signature`	2	A permission that the system is to grant only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.
`signatureOrSystem`	3	A permission that the system is to grant only to packages in the Android system image *or* that are signed with the same certificates. Please avoid using this option, as the signature protection level should be sufficient for most needs and works regardless of exactly where applications are installed. This permission is used for certain special situations where multiple vendors have applications built in to a system image which need to share specific features explicitly because they are being built together.

Figure 7. Custom Permission Protection Levels. The developer must choose one of these levels for his custom permission.

applications just to certain sites), while an application without the INTERNET permission cannot directly access the network at all.

5.2.3 Flows Among Applications

The final type of information flow we consider is flows from one application to another. As we mentioned in Section 3.2, components of an application (activities, services, content providers, and broadcast receivers) can communicate with one another through IPC by sending intents. The manifest.xml file specifies whether or not each component of an application can be invoked from other applications, and what permissions, if any, are required by the calling application. We explain this in greater detail.

The <activity>, <service>, <provider>, and <reciever> tags all contain a field called exported. If this is set to true, then the component is accessible to external applications. If this is set to false, then the component can only be invoked from within the application. Alternatively, the exported field can be left unspecified, and the interpreter will infer whether or not the component should be externally accessible based on the presence of intent filters: if that activity, service, provider, or receiver specifies intent filters then the component is assumed to be externally accessible, but if there are no intent filters then the component is assumed to be accessible from only within the application.

Those components that can be invoked by other applications may or may not have permissions associated with them (but if they involve revealing or manipulating sensitive data, they should). Permissions for an individual component are specified in the permission field of the component's tag. In order for an application to invoke a permission-protected component of another application, it must hold those permissions.

Consider a hypothetical online shopping application. Suppose that application has an exported activity that, when invoked, returns to the caller the users shopping list. If this activity were not protected, then any application could call it and access the user's shopping list. However, if the developer specifies a READ_SHOPPING_LIST for the activity, then only applications with the READ_SHOPPING_LIST permission would be able to invoke this activity.

5.3 Flows Outside the Model

Since Android is built on Linux, and applications are really just Linux processes, exploits in the Linux IPC model could potentially lead to security threats in Android. Furthermore, it has been suggested that it might be possible for applications to communicate with one another on the Android platform outside the permission/API model through world-writable files. Neither of the two applications we examined in detail communicated with world-writable files, so we do not concern ourselves with this in our analysis.

6. Case Study #1

For the first case study, we considered the k9mail application. This application is an open source email client written in Java. It has over 250,000 users and due to its nature, this application has access to lots of private data including contact information, personal calendars, and email contents.

Because this application is open source, we were able to download all of its Java files and examine them in Eclipse. The analysis was done by hand, due to the automated tools and infrastructure not being available. The main goal of the analysis was to determine how information flows through the application, and to make sure that there are no flows of private data beyond what is needed for a functional email client.

In performing our analysis, we directly apply the model from Section 5. We first consider the sources of private information in k9mail, specifically how K9mail collects sensitive user information and which sensitive APIs it has permission to access. Then, we consider how information can flow from k9mail to the network (since k9mail has the INTERNET permission). Finally, we look at at what data k9mail can send and receive from other applications through IPC (in particular through its services, broadcast receivers, and activities).

6.1 Sources of Private Data

We begin by considering what private data k9mail is able to collect. As explained in Ssection 5.1, the three sources of private data are Android hardware, default software and applications, and data that users direclty provide to 3rd-party applications.

The first two of these three sources are protected by the Android API, and the corresponding default Android permissions. In Table 1 we provide all the default permissions that k9mail requests (from the <uses-permission> tags of the manifest.xml file). In particular, we consider the read permissions, since these are permissions that allow k9mail to access protected APIs. These permissions are straight-forward, and are all things a functional email client would need (the user's contacts, the network state, etc).

In our examination of the code, we found no evidence of any of these being used inappropriately. It is also worth mentioning that although k9mail can write to an SD card (it has the

Default
RECEIVE_BOOT_COMPLETED
READ_CONTACTS
READ_SYNC_SETTINGS
READ_OWNER_DATA
GET_ACCOUNTS
WRITE_CONTACTS
ACCESS_NETWORK_STATE
INTERNET
VIBRATE
WAKE_LOCK
WRITE_EXTERNAL_STORAGE
READ_KEY_DETAILS
Custom
READ_ATTACHMENT
READ_MESSAGES
DELETE_MESSAGES
REMOTE_CONTROL

Table 1. Permissions declared by the k9mail application. Note that this application requests permission to change the user's contacts, access the network state, and modify the phone's settings. It also declares four custom permissions.

WRITE_EXTERNAL_STORAGE permission) it cannot read from an SD card, so we do not need to worry about it accessing private information in that manner.

The third potential source is data that users input directly into the application via some form. By looking at the code and playing with the application, we concluded that the only private information k9mail can directly collect is new contact data, the contents of emails that users write, and more passive information about user behavior (when they send/open emails, what their k9mail settings are, etc).

In the first two cases, k9mail has (and needs) the READ_CONTACT_DATA and WRITE_CONTACT_DATA permissions. Given this, the fact that it collects contact data and sees email messages is not concerning (we consider in the next section if this data is sent anywhere it should not go over the network). In the third case, we saw nothing to indicate that K9mail was recording a user's behavior.

6.2 Network Flows

As an email client, k9mail requests the INTERNET permission which allows it to talk to the network. In this section we consider how information flows to the network. In particular, we want to be confident that nothing beyond user-submitted email messages get sent.

Only one file in this application connects to the network by means of an API call. This call returns the network's state and connectivity status and some other coarse network information. An email is then pushed to the network or pulled from the network through this connection. There are no other interactions with the network present in this application – the network activity is modularized to only the MailService.java file.

6.3 Flows with Other Applications

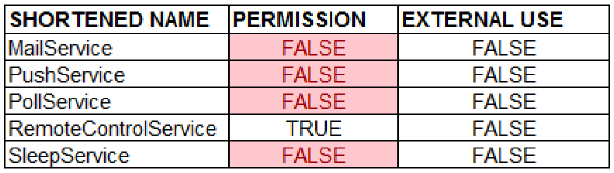
In this section we consider potential flows between K9mail and third party applications. We break our analysis down into flows through services, flows through broadcast receivers, and flows through activities. As mentioned in Section 3.2 these components of applications can communicate with components of another applications by sending intents. We consider how, if at all, private user data could flow through any of these components.

6.3.1 Flows Through Services

Analyzing flows from k9mail services was very straight-forward. We got a list of all the k9mail services by examing the manifest.xml file, and then considered if other applications could invoke any of them.

As we mentioned in Section 5.2.3, a service can be invoked externally if either the exported field is set to true, or if intent filters are provided. For all the k9mail services, the exported field is set to false (what it defaults to) and no intent filters are specified. Therefore, none of these services can be called outside k9mail.

In Figure 8 we list all the services, noting that none of them can be externally invoked. For thoroughness, we also specify which have permissions protecting them, but this is not particularly relevant for our analysis of flows between applications, since no external applications can invoke any of these services.



SHORTENED NAME	PERMISSION	EXTERNAL USE
MailService	FALSE	FALSE
PushService	FALSE	FALSE
PollService	FALSE	FALSE
RemoteControlService	TRUE	FALSE
SleepService	FALSE	FALSE

Figure 8. Information Flow Among the Services in k9mail.

6.3.2 Flows Through Broadcast Receivers

We approached our analysis of possible flows through broadcast receivers in a similar manner. We obtained a list of all broadcast receivers from the manifest.xml file, and then for each one considered (1) if it is permission protected and (2) where exactly this broadcast receiver can receive messages from. Determining if a broadcast receiver is permission protected is straightforward, since permissions are just specified in the <receiver> tag of the manifest.xml file.

Determining the sources (system broadcasts or broadcasts from other applications) that invoke each of these receivers was a little more involved. As mentioned in Section 3.2, applications send and receive communications through intents, with intent filters specifying which components of which applications should receive which intents.

Accordingly, we examined the intent filters for each of the broadcast receivers. Android has specific filter rules, which are discussed in full detail in the Developer Guide [2]. With these in mind, we analyzed each of the intent filters for all of the receivers to determine how they could be invoked. Although the rules for resolving intents can be intricate, in the case of k9mail all the receivers respond to either system broadcasts, or by-name intent calls from other applications.

We summarize this analysis in chart Figure 9. For each of the recivers (there were only 4) we list all the system broadcasts, as well as the external application broadcasts, that can send messages to each receiver, and specify whether or not the receiver is protected by a permission.

Two of the receivers, CoreReciever and StorageReceiver, only respond to system broadcasts, so there is no need to worry about external applications invoking them. RemoteControlReceiver can (and is meant to) receive and respond to requests from external aplicaions. However, it is permission protected by a custom k9mail permission (REMOTE_CONTROL), so only applications that have explicitly been granted permission by the user may communicate with the RemoteControlReceiver.

Finally, we consider the BootReceiver receiver. It responds to several broadcasts, all but one of which are system broadcasts. The non-system broadcast, scheduleIntent, can be invoked by third-party applications. Furthermore, BootReceiver is not protected by a permission. We examined the source code closely to determine if this could lead to any undesirable information flows.

As it turns out, scheduleIntent allows applications to send appointment reminders to k9mail which pop up as alarm alerts. No information is sent back to the calling application, beyond a RESULT_OK message (which only reveals that the intent was received). At worst, a malicous application could exploit this to send k9mail lots of annoying appointment reminder messages, but a user could easily identify this and remove the malicous application, and there is no flow of private information.

Based on this analysis, we conclude that no inter-application communication between k9mail broadcast receivers and external applications can lead to leaks in private user information.

6.3.3 Flows Through Activities

Our analysis of possible flows through activities began with an examination of the manifest.xml file. We obtained a list of all activities, and then for each one considered if it could be invoked externally at all (the presence of intent filters indicates whether or not an activity can be called externally). If so, we determined what information, if any, k9mail can send to or received from the application that invoked it. We did this by examining the relevant API calls in the corresponding java files for each activity.

This is summarized in Figure 10. Of the 24 activities only 6 can be invoked externally. For brevity, we omit most of the activities that can not be externally invoked in chart Figure 10. Also, since none of the activities were permission protected, we omit this column from the chart.

Three of the 6 activities that can be called externally neither send data to nor receive data from the application that calls it (Accounts, FolderList, and Search). This means that another application is allowed to directly jump to the activity (such as the accounts list page) in k9mail, but once the jump happens no further communication happens between the applictions. For example, any application could send an intent to k9mail asking it to open the list of accounts, but no additional information flows from the calling application to k9mail, and no information flows back from k9mail to the calling application (none of the contact information, for instance, is sent back).

Of the remaining 3 activities that can be called externally only one is capable of sending information back to the invoking application (LauncherShortcuts). After a close inspection of the java file, we determined that the only piece of information that LauncherShortcuts can send back to the calling application is a RESULT_OK message, a one-bit flag that indicates whether or not the intent succeeded.

The last two activities that can be called externally are capable of receiving information from (but not sending information to) applications that invoke them (MessageView and MessageCompose). In both cases the information that flows to k9mail is the contents of an email (either the message to be viewed, or a partially composed email). After carefully examining the code for these activities, we conclude that k9mail is not recording, tracking, or doing anything else nefarious with the contents of a user's emails (k9mail simply displays the messages and sends them to the intended recipients when the user clicks send).

A threat present with any email viewer is the loading of remote content into an email message, where a malicious application embeds private data in the URL. Before loading such content, k9mail asks for the user's permission. If users naively ignore this warning then it is certainly possible for malicous applications without the

ABBREVIATED NAME	PERMISSION	SYSTEM BROADCASTS	OTHER BROADCASTS
BootReceiver	FALSE	BOOT_COMPLETED	
		DEVICE_STORAGE_LOW	
		DEVICE_STORAGE_OK	
		CONNECTIVITY_CHANGE	
		BACKGROUND_DATA_SETTING_CHANGED	
		SYNC_CONN_STATUS_CHANGED	
			com.fsck.k9.service.BroadcastReceiver.scheduleIntent
RemoteControlReceiver	TRUE		com.fsck.k9.K9RemoteControl.set
			com.fsck.k9.K9RemoteControl.requestAccounts
CoreReceiver	FALSE	WAKE_LOCK_RELEASE	
StorageReceiver	FALSE	MEDIA_MOUNTED	

Figure 9. Information Flow Among the Broadcast Receivers in k9mail.

INTERNET permission to exploit this to send private user data to servers. However, this threat applies equally to the default Android email viewer, and is a potential problem for email clients in general.

Based on this analysis, we conclude that k9mail sends no private data to other applications (aside from the very low information RESULT_OK bit). Furthermore, K9mail only receives information from other applications in the form of email messages, and it does not seem to do anything abusive with the contents of emails. Finally, k9mail takes the standard precaution of requesting users consent before loading remote content.

ABBREVIATED NAME	EXTERNAL USE	RECIEVES DATA	SENDS DATA
Accounts	TRUE	FALSE	FALSE
FolderList	TRUE	FALSE	FALSE
MessageView	TRUE	TRUE	FALSE
MessageCompose	TRUE	TRUE	FALSE
Search	TRUE	FALSE	FALSE
LauncherShortcuts	TRUE	FALSE	TRUE
s.Prefs	FALSE		
s.FontSizeSettings	FALSE		
s.AccountSetupBasics	FALSE		
s.AccountSetupAccountType	FALSE		
s.AccountSetupIncoming	FALSE		
s.AccountSetupComposition	FALSE		
s.AccountSetupOutgoing	FALSE		
...	...	...	...

Figure 10. Information Flow Among the Actitivities in k9mail.

7. Case Study #2

For our second case study we analyzed an application called Talking Cat, a knock-off of a very popular application called Talking Tom Cat. The application is an entertainment application that consists of a virtual cat that the user can interact with through the touch screen. It will also repeat (in a "funny voice") anything the user says into the devices microphone.

Although not as popular as Talking Tom Cat, Talking Cat still has over 500,000 users. In addition to be popular, we thought it was a valuable application to analyze since it requests both the ACCESS_COARSE_LOCATION and INTERNET permissions (neither of which should be necessary for a talking cat application). Talking Cat was much smaller and more straightforward than k9mail, which made our analysis of it much simpler.

7.1 Decompilation

Unfortunately, Talking Cat is not an open-source application, so we first had to find a way to decompile the code before we could analyze it. We used a tool called dex2jar [3] that converted the byte code into human-readable java (although it was obfuscated), but left the crucial manifest.xml file in binary form. To read this we used a tool called AXMLPrinter2 [4], which converted it back into a text file.

7.2 Sources of Private Data

As with the k9mail case study, we begin by considering what private data Talking Cat is able to collect. As explained in Section 5.1, the three sources of private data are Android hardware, default software and applications, and data that users directly provide to 3rd-party applications. The first two of these three sources are protected by the Android API, and the corresponding default Android permissions.

In the default column of chart Table 2 we provide all the permissions that Talking Cat requests (from the uses-permission tags of the manifest file.xml file). We note that the application has access to the users microphone (RECORD_AUDIO), location data (ACCESS_COURSE_LOCATION), and the network (INTERNET). The first of these three is necessary for the function of the application, but it is unclear why a virtual talking cat would need to access the user's location data or interact with the network. When we consider network flows and flows between applications in the next two sections, we examine how this unneeded private data can be shared.

The other potential source of private information is data that users input directly into the applications (usually by inputting information into forms). By looking at the code and playing with the application we concluded that Talking Cat is not collecting private user data in this manner.

7.3 Network Flows

To determine how Talking Cat communicates with the network we scanned through every file that communicated with the Network API. We determined that the application communicated with the network only in the context of Google's Ad frameworks.

Talking Cat accessed the network in one file – AdUtil.java. This file uses the network API to fetch the type of the network and its

Default
ACCESS_COARSE_LOCATION
ACCESS_NETWORK_STATE
INTERNET
WAKE_LOCK
WRITE_EXTERNAL_STORAGE
RECORD_AUDIO
READ_PHONE_STATE

Table 2. Permissions declared by the Talking Cat application. Note that this application requests access to the internet, location data, and the network.

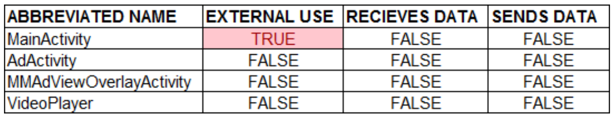
state. It then uses the user's GPS location to pull in an add from the add server. The add is finally displayed in a separate view.

It is clear that the application declares the INTERNET and the COARSE_LOCATION permission with the explicit intent of displaying targeted adds to the user. There is no other use of the internet connection or the user's location in Talking Cat. This targeting is done via Google Ads, and we constrain our analysis to determining that this application does interact with that server in one single file.

7.4 Flows with Other Applications

Unlike k9mail, Talking Cat is largely isolated from communication with other applications. It has no services or broadcast receivers, just activities (this is not surprising given that it is just a simple interface). In an analogous manner to our k9mail analysis, we looked through the permissions and intent filters of all the activities in the manifest.xml file. None of the activities were permission protected.

However, only a single activity was capable of being invoked externally. This activity, MainActivity, is the launcher for the application. We then inspected its java file (looking for the relevant API calls) to determine if MainActivity can receive any data from an application that calls it, or if MainActivity can send any data back to a calling application. It turns out that MainActivity can do neither of those things. An external application can only launch Talking Cat, and do nothing more. This is summarized in Figure 11.



ABBREVIATED NAME	EXTERNAL USE	RECIEVES DATA	SENDS DATA
MainActivity	TRUE	FALSE	FALSE
AdActivity	FALSE	FALSE	FALSE
MMAdViewOverlayActivity	FALSE	FALSE	FALSE
VideoPlayer	FALSE	FALSE	FALSE

Figure 11. Information Flow among the actitivities in Talking Cat.

8. Discussion

In light of our foray into the realm of Android Security, we are convinced that in order to analyze applications effectively automated analysis is necesssary. This automated analysis needs tackle implicit and explicit flows to provide the user with guarantees about the security of their private data.

The smart phone platform presents some unique challenges for information flow tracking. For one, these platforms have less computing power and are much more sensitive to high overhead costs. Additionally, smartphones and their interactions with applications present different types of sensitive information to consider.

These sources of information need to be treated differently and any sort of automated analysis would need to account for this. Further, because applications can share information, information flow analysis cannot be done at the application level. It must be more fine grained, however, the more fine grained it gets the higher the risk of a high overhead cost.

Acknowledgments

We'd like to acknowledge Prof. Stephen Chong for his guidance in this project. We would also like to acknowledge the CS252r Spring 2012 class for their helpful comments during our project presentations.

References

[1] Android security overview, . URL `http://source.android.com/tech/security/index.html`.

[2] Security and permissions, . URL `http://developer.android.com/guide/topics/security/security.html`.

[3] Almost half of u.s. smartphones running android, . URL `http://code.google.com/p/dex2jar/`.

[4] android4me, . URL `http://code.google.com/p/android4me/`.

[5] In-stat: Majority in u.s. to have smartphones, . URL `http://news.cnet.com/8301-13506_3-20095949-17/in-stat-majority-in-u.s-to-have-smartphones-tablets-by-2015/`.

[6] Almost half of u.s. smartphones running android, . URL `http://www.pcmag.com/article2/0,2817,2401250,00.asp`.

[7] 1 in 5 android apps pose potential privacy threat, . URL `http://mashable.com/2010/06/23/android-apps-privacy-threat/`.

[8] J. Burns. Moblie application security on android. *Black Hat USA*, 2009. URL `http://www.blackhat.com/presentations/bh-usa-09/BURNS/BHUSA09-Burns-AndroidSurgery-PAPER.pdf`.

[9] 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 real-time privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924971`.

[10] W. Enck, D. Octeau, P. Mcdaniel, and S. Chaudhuri. A study of android application security. *USENIX Security*, (August):935–936, 2011. URL `http://www.usenix.org/event/sec11/tech/slides/enck.pdf`.

[11] J. Jeon, K. K. Micinski, J. A. Vaughan, N. Reddy, Y. Zhu, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. Technical Report CS-TR-5006, Department of Computer Science, University of Maryland, College Park, December 2011.