Graphical Representations of Security Settings in Android

by

Aaron Gibson

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2015 by the
Graduate Supervisory Committee:

Rida Bazzi, Chair
Gail-Joon Ahn
Erin Walker

ARIZONA STATE UNIVERSITY

May 2015

ABSTRACT

On Android, existing security procedures require apps to request permissions for access to sensitive resources. Only when the user approves the requested permissions will the app be installed. However, permissions are an incomplete security mechanism. In addition to a user's limited understanding of permissions, the mechanism does not account for the possibility that different permissions used together have the ability to be more dangerous than any single permission alone.

Even if users did understand the nature of an app's requested permissions, this mechanism is still not enough to guarantee that a user's information is protected. Applications can potentially send or receive sensitive information from other applications without the required permissions by using intents. In other words, applications can potentially collaborate in ways unforeseen by the user, even if the user understands the permissions of each app independently.

In this thesis, we present several graph-based approaches to address these issues. We determine the permissions of an app and generate scores based on our assigned value of certain resources. We analyze these scores overall, as well as in the context of the app's category as determined by Google Play. We show that these scores can be used to identify overzealous apps, as well as apps that do not properly fit within their category. We analyze potential interactions between different applications using intents, and identify several promiscuous apps with low permission scores, showing that permissions alone are not sufficient to evaluate the security risks of an app. Our analyses can form the basis of a system to assist users in identifying apps that can potentially compromise user privacy.

To Scott, Cheryl, and Jordan Gibson

ACKNOWLEDGEMENTS

I would like to thank Professor Rida Bazzi for his mentoring and direction, especially when my work needed focus. I would also like to thank my committee of Professor Gail-Joon Ahn and Professor Erin Walker for their advice and feedback. I'm very grateful to Professor Yau and the Scholarship for Service (SFS) program for sponsoring my studies and research. I would also like to thank many friends for their continued support; in addition to their important feedback, their humor, reassurance, and friendship will forever be treasured. I would like to thank my family, who provided strength and comfort throughout this whole process and my life. Finally, I would like to thank God, for which this was all possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Mobile devices are becoming increasingly widespread and pervasive. The availability of frameworks and SDKs for these devices make it easy for developers to create apps for these devices. However, the ubiquitous nature of these mobile devices allows them to collect more sensitive information about users; this makes them more lucrative to attackers and other third parties, notably the app developers themselves.

As a result, Android comes with built-in security mechanisms. Individual apps are sandboxed[1] and cannot access the data of the other applications. Apps can only access sensitive system resources by requesting the proper permissions at install time. Finally, apps can still send data between each other by sending *Intents*.

However, these mechanisms have certain limitations. First, an application can only be installed if the user accepts all of the permissions requested by an application. This all-or-nothing approach may encourage a user to just accept everything, which hinders the protection offered by permissions in the first place. Furthermore, many users will not understand all of the permissions requested by an application and will be inclined to accept out of ignorance Felt *et al.* (2012). As a result, users may not know of the data that their downloaded applications collect. For example, a seemingly benign wallpaper app that collects information such as their device ID and subscriber ID[2], much to the chagrin of the user Enck *et al.* (2014); does a wallpaper application

[1] Note that apps signed by the same certificate can access each other's data if configured to do so. In any case, apps signed by the same certificate are from the same developer.

[2] The subscriber ID is a secret ID assigned by the phone company to identify the phone; an attacker that knows the subscriber ID can impersonate the user.

really need to know the user's subscriber ID?

Second, applications can send or receive sensitive information from other applications even if they do not have the required permissions through the Intent mechanism. For example, an app that does not request access to the internet can still send sensitive data over the internet by constructing an intent with a custom URL attached; when the Android device processes the intent, it might open that URL, which can leak data via GET requests Qiu (2012). It is also possible for malicious apps to register to accept any type of intent, leaking information to the app.

The more general problem is that the user is largely unaware of the information that many apps collect, especially outside of their perceived context. Borrowing again from the example in Enck *et al.* (2014), a user should not expect a wallpaper app to necessitate the phone's subscriber ID; after all, displaying wallpaper has nothing to do with the cellular provider. Yet, users have installed the application anyway, most likely ignoring the required permissions. It is likely that these users are completely unaware that they are sending sensitive cellular information, even though they granted the app access to the subscriber ID in the first place.

Some existing work targeting Android suggest different techniques to identify malware or malicious activity. Banuri *et al.* (2012) and Holavanalli *et al.* (2013) both focus on detect malicious usages and potential information flows created from Android permissions. While useful, these are used to detect malware, and not other potential privacy violations by legitimate, but promiscuous adware apps. Other work targets the leakage of permissions by using intents, such as ComDroid Chin *et al.*

(2011).

1.1 Motivation

As our motivating example, consider a popular Flashlight app for Android, called *Super Bright LED Flashlight*[3]. This app has millions of downloads and over 4.4 million reviews Surpax Technology Inc. (2015). It requests the following system permissions, which grant the app access to the corresponding system resource[4]:

- CAMERA
- FLASHLIGHT
- CHANGE_CONFIGURATION
- WRITE_SETTINGS
- WAKE_LOCK
- ACCESS_NETWORK_STATE
- INTERNET

Given that the flashlight app might possibly use the LED for the camera's flash, these permissions are not very aggressive. While it is curious that this app is requesting access to the Internet (probably for ads), the app does not request much else that is sensitive, suggesting that users might view this app as safe. As evidence, some of the top reviews for this app are listed in Table 1.1. However, as we shall see later, this

[3] This app is different from the Flashlight app considered by Felts et. al. (Felt *et al.*, 2012); it would be interesting to know whether this work motivated the less-intrusive permissions requested by the Super Bright LED Flashlight app.

[4] The full name of these permissions are usually prefixed with *android.permission*. We borrow the convention in many academic papers on Android permissions, where this prefix is dropped and inferred unless otherwise stated.

Rating	Comment
5	**No doubt, i like it** Its said require camera or flash hardware...but my advice havn't...morethan its used usefull...tq so a lot.
1	**Im uninstallin** Just heard a report bout how everytime u turn it on ppl n other countrys use that info its a spy kinda. confirmed is russia and china and I highly recomend readin the terms b4 nstallin cuz it supposibly states that it spys cuz they had to state it cuz they were sued
5	**Does what it does.** The people saying this spies: it needs camera access to use the flash. Microphone and camera access are lumped under one permission. There are no other permission, such as contact list or browsing history. It's not spying.

Table 1.1: This table lists some of the first reviews displayed on the Google Play store for this app (Surpax Technology Inc., 2015). Out of the six reviews on the first page, five reviews gave the app five stars, while one gave the app one star. We have chosen to display these specific reviews as they talk about the app's requested permissions or security risks. The reviews were copied directly over, including any misspellings.

flashlight app has the ability pass information to other apps to do things like send email, compose SMS messages, and make phone calls, even though it never requests those particular permissions.

1.2 Goals

Our goals in this thesis are to examine how collaboration between different resources and applications can still result in security threats. We will demonstrate a scoring on permissions that analyzes potential privacy violations due to collaboration of the requested permissions. Then, we will demonstrate promiscuous information flows that are possible between seemingly benign Android apps, some of which do not request aggressive permissions. The remainder of this paper is structured as follows. First, we will describe our model for security. Second, we will describe the Android ecosystem and its existing security mechanisms. Next, we will apply our model to this ecosystem, then present the results of our work. The last chapter concludes the paper.

Chapter 2

RELATED WORK

Existing research in this area largely targets Android permissions, which indicate specific resources that a particular app will have access to. Banuri et. al. suggest detecting sequences of invoked permissions in order to identify malicious flows Banuri *et al.* (2012). Holavanalli et. al. create a static analysis tool, Blue Seal, to detect potential information flows created from Android permissions Holavanalli *et al.* (2013). Ongtang et. al. propose improving Android permissions by allowing for encoding runtime policies for them Ongtang *et al.* (2009).

Additional work has been done in detecting the leakage of permissions via intents. Sbirlea et. al. demonstrated that some applications leak their access to sensitive, permission-protected resources via intents due to poor configuration Sbirlea *et al.* (2013). Chin et. al. develop a static analysis tool to analyze the intents sent out by an application and identify whether it matches what is advertised via the registered intent-filter Chin *et al.* (2011). This tool was designed to aid developers in making their application more secure against malicious or incorrect usage of the different components it adds to the device. This work is complementary to ours; while they focus on securing intents within a single app, we focus on whether different apps can intercommunicate.

Other solutions have considered information flow in a more general context. Enck et. al. augment the Android runtime framework to append taints to data as it flows through the system. The taints store origin and some provenance information about each bit of data, demonstrating that even seemingly benign applications send out data that the user does not expect Enck *et al.* (2010). Tiwari et. al propose a reworking of the Android API to split the data for each user into light-weight contexts, which they call bubbles. Each bubble models a specific context where application data can be stored. This proposal requires a reworking of the Android API to grant users the ability to identify which apps belong where Tiwari *et al.* (2012).

Complementary work has been done on improving the Android sandbox. For example, Xu *et al.* (2012) creates a tool, Aurasium, that unpackages an Android app, then repackages it with custom hooks that allow for tracking the apps behavior and stopping any potential malicious activity Xu *et al.* (2012). This work is complementary to ours, since their custom sandbox still requires a configuration of desired security policies.

Chapter 3

ANDROID OVERVIEW

Before describing the approach, we provide an overview of the Android ecosystem. The Android OS is built on top of special builds of the linux kernel, with custom drivers and libraries to cater to the needs of mobile platforms. Many of the common features in the linux kernel are removed, such as interprocess communication techniques (Section 3.3), since they incur overhead that does not fit the requirements of the mobile platform.

3.1 Structure of an Application

Each Android application is written in Java that links against the Android SDK. These applications are typically compiled into Java bytecode, then recompiled into Dalvik bytecode[1]. It is also possible for Android apps to contain native code written in C/C++ through the Android NDK, but this is discouraged by Google unless absolutely necessary for performance, such as for gaming apps. Even apps written in the native NDK require a hook into the Android framework to load the native code, which is written in Java.

Each Android application is distributed as an APK (Android Package) file, which is a special Zip archive file that bundles the code, images, and other similar resources. This Zip file is signed by the developer using their private key for their certificate,

[1] Dalvik is a special Java virtual machine that is optimized for embedded environments. Newer apps for Android 5.0 may use another, faster Java virtual machine called ART (Android Runtime), though this is not yet mainstream Google (2015b). This difference does not affect any of the analysis in this paper.

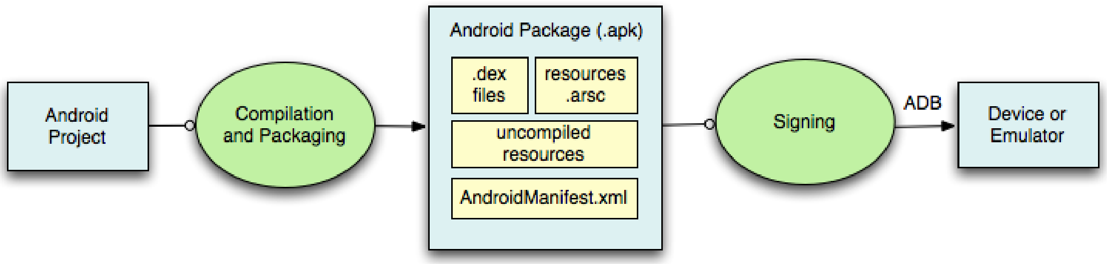


Figure 3.1: This figure outlines the build process for Android applications as adapted from Google (2015b). Note that the application must be signed in order to be released to the Google marketplace.

and contains the following:

- **AndroidManifest.xml** – This binary XML file contains much of the meta-data for the Android application, such as the specific components that this application contains, and intent-filters (explained further in Section 3.3).
- **X.509 certificate** – This stores the public key and certificate of the developer in PKCS#7 format. It is important to note that these certificates are *self-signed*.
- **classes.dex** – This contains the actual code for the application. If the application also uses native code compiled from C/C++, then those libraries are also included within the zip archive.
- **Resource XML files** – These files store the application's resource information, such as how to layout components on different screens, what strings to display depending on locale and language, the locations of icons and images, and any look and format information for UI elements.

The build process for Android applications is shown in Figure 3.1. Each Android application is installed as a separate user within the linux kernel; this is what effectively creates the application sandbox with limited access to system resources[2].

3.2 Permissions

For an app to access additional, usually sensitive system resources, it must request the corresponding *permissions* in its AndroidManifest.xml file. Permissions are implemented in Android as user groups in linux; applications can only access the respective resource if they are members of the proper user group. As such, the process of adding apps to specific groups happens during install-time, and the permissions cannot be removed or changed without reinstalling or updating the app. When installing an app, the user may be prompted to check and approve of the app's requested permissions; any *dangerous*[3] permission must be explicitly approved by the user before installing.

As an example, if an application wishes to access the camera on the Android device, they must request the CAMERA permission. Since the CAMERA permission is deemed *dangerous* by the Android ecosystem, a user will be prompted whether this permission is okay for this application during install time. If the user declines (i.e. does not grant the app the CAMERA permission), the app will not be installed.

3.3 Intents

Android does not allow for using the standard linux IPC mechanisms. Instead, Android uses Binder, a custom driver, to share information between applications. The Binder driver is optimized for embedded devices because it allows for more efficient

[2] It is possible for several different applications to exist in the same sandbox if the apps request as such and both apps are signed with the same developer certificate Google (2015b). We do not consider this case because it is likely that two differing apps by the same developer could exchange information anyway outside of the device.

[3]The sensitivity of the permission, i.e. whether it is dangerous, is specified when it is declared. The sensitivity of system-defined permissions are documented at Google (2015b).

Component Type	Description
Activity	A component that can interact directly with the user (i.e. a GUI component).
Service	A component that performs work in the background.
Broadcast Receiver	A component that respond to system notifications.
Content Provider	A component that provides access to system and application data.

Table 3.1: This table lists the Android components that can receive intents. Applications define their own components by subclassing one of these types offered by the Android SDK.

sharing of resources and file descriptors between apps. If an app wishes to transfer a photo to another app, for example, then the app can transfer the file descriptor instead of the whole file, saving the additional overhead of maintaining a copy of the file in a separate context. The Binder driver also allows for Remote Procedure Calls (RPC), an important use case for the component-driven architecture that Android uses.

To abstract away the complexities of using Binder directly, the Android ecosystem defines *Intents* which encapsulate the parcels of data to be sent over the Binder driver to different applications. Intents can be used to spawn the types of Android components listed in Table 3.1.

An intent can contain the following fields:

- Action – (*Required*) A string representing the action to perform, or the target component's fully scoped package name.
- Category – The category of this intent.
- Data – A field that indicates the MIME type or URL of the data.

- Extras – A set of key-value pairs with additional metadata for the intent.
- Flags – A bitmask of options to customize how the Android framework resolves the intent.

Intents that contain the target component's package name are called *explicit intents*, whereas intents with an action string, such as android.intent.action.VIEW are called *implicit intents*. An intent is sent to the system to be resolved by a call to the appropriate method, such as[4]:

- startActivity()
- startService()
- bindService()

These calls are accessible through the base classes of Android components provided by the framework. We note that even native code requires using Java calls to the system through the JNI interface[5].

In order for applications to receive Intents, they should generally specify the kinds of intents they can receive. This is done by declaring an intent filter in their AndroidManifest.xml file. The intent filter *must* declare the action it can handle, as well as some optional additional information to filter what it receives, notably the allowed categories and the allowed data for the intent. Figure 3.2 shows an example of an intent filter for an activity within a sample Android app. Note that the intent filter

[4] Note that this list is by no means exhaustive.

[5] Recall that the Android ecosystem still requires certain hooks to be in place for the apps to register with the system. These hooks require interfacing with the Java portion of the framework, even if the app itself is otherwise written in another language like C++. This requirement is leveraged by other tools as well, such as Aurasium Xu *et al.* (2012), to ensure the full security of their sandboxes.

```
<activity class=".EditMediaActivity" exported="true">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
    <data android:mimeType="video/*" />
  </intent-filter>
</activity>
```

Figure 3.2: A sample intent filter in the AndroidManifest.xml file. This example is based on the developer documentation by Google (2015b).

,

can declare multiple actions, categories, and data that it can respond to in the same filter.

The Android framework determines the possible component that an implicit intent can be resolved with as follows:

1. Check whether the given Android component is exported. If not specified explicitly, the component is assumed to be exported.

2. Check whether the intent action matches any of the listed actions in the intent filter. The intent must match one of the actions listed in the filter in order to pass this check.

3. Check whether the intent category matches any of the listed categories in the intent filter. Only intent-filters that allow for the android.intent.category.DEFAULT are allowed to handle implicit intents. If the intent-filter contains this category, the every category contained in the intent itself must also match a category

listed in the filter to pass. If the intent does not contain any categories to match, then it passes this check by default.

4. Check whether the intent matches any of the data parameters listed in the intent filter. Resolving the data type here is more involved since the data tag can contain a URL or a MIME type.

Android apps can also receive explicit intents if they are referenced directly by package name; direct access from another application can only be avoided by setting the *exported* attribute false.

Chapter 4

GRAPHICAL REPRESENTATIONS OF SECURITY SETTINGS

Android applications are inherently isolated by the linux kernel Google (2015b). Thus, the security of the Android ecosystem depends heavily on the information flows that are possible in the system. We forumulate our graph-based approach with this observation.

4.1 Model Formulation

Motivated in part by the graph-based access control model from Xu et. al. Xu *et al.* (2009), we define our abstract model as follows: Systems comprise of a set of *objects* V, each with some state. Given two objects, $v_i, v_j \in V$, they can interact in the following ways:

- $write(v_i, v_j)$ implies that v_i can write to v_j.
- $read(v_i, v_j)$ implies that v_i can read the state of v_j.

For the interactions above, the details of the objects are not relevant; it is only necessary that these objects have some notion of *read* or *write*[1]. We define I_r and I_w as the set of all possible read and write operations in the system respectively:

$$\begin{aligned} I_w &= \{(v_i, v_j) : write(v_i, v_j) \text{ is possible in the system.}\} \\ I_r &= \{(v_i, v_j) : read(v_i, v_j) \text{ is possible in the system.}\} \end{aligned}$$

[1] The paper from Xu *et al.* (2009) assumed a more sophisticated model involving processes, files, and other objects on and between computers. The model that we present here is intentionally more abstract, noting that such differences have little theoretical significance in the current context.

We define a directed graph, $G = (V, E)$ with the edges defined as follows:

$$E = \{e_{ij} : (v_i, v_j) \in I_w \text{ or } (v_j, v_i) \in I_r\}$$

The element e_{ij} implies the directed edge from $v_i \to v_j$. Note that the model above makes no other assumptions about the properties of a given edge e_{ij}.

4.2 Application to Android Permissions

In order to apply the model above to Android permissions, we note that permissions protect specific resources on the user's device. We first make some observations about the types of resources that permissions protect in Android.

Permissions can be used for arbitrary resources on the system [2], but they generally protect two types of resources:

1. Sensitive Local Resources

2. Access to Remote Resources

For example, the READ_CONTACTS permission protects access to the database with the user's contact information, and thus protects a *Sensitive Local Resource.* The INTERNET permission, however, protects access to the creation of network sockets, and thus protects *Access to a Remote Resource.* Both types of resources have the ability to be written to or read from and thus fit into the graph model outlined in Section 4.1.

If we assume that an application is self-contained, i.e. does not send data via Intents to other installed applications on the device, then the possible information flows between resources in the application depend on what permissions the application

[2] Note that applications can actually create their own permissions, if desired. In these cases, the resource in question is generally one provided by the application itself.

requests. A cautious reader may note that this assumption is not likely to be valid for all apps, a point which we shall discuss later in Section 4.3.

For an isolated application (one that does not communicate with other installed apps), we observe that there two subjects that the app can still potentially communicate with:

1. The user himself.

2. The world.

The user is bidirectionally connected directly to every local resource the app implicitly requests through its permissions; this is due to the fact that each local resource is really some subset of the total information for the user, for which the user can read and write. In a similar manner, every transit resource is bidirectionally connected directly to the world.

When an application requests a permission that grants *read* access to a local resource (such as READ_CONTACTS), we construct an edge from that local resource to all transit resources the application uses. This is because the application can read that resource and send it out via any transit resource it has access to. When an application requests a permission that grants *write* access to a local resource, we construct an edge from all transit resources the application uses to that local resource. Note that it is possible for a single permission to imply both read and write access, such as the CAMERA. When an application requests a permission that protects a transit resource, we add that node to the available transit resources; note that transit resources are inherently bidirectional and the permissions that protect them imply both read and write access.

The resulting graph generated above maps out the possible information flows for an isolated application, but the details and implications for particular flows are not yet considered. For instance, an email application might request a user's contacts in addition to the internet. There are countless interactions that these particular resources alone might legitimately use, such as[3]:

- Finding the email address in the contact list to send a new message.
- Sending a reply to some received message.
- Creating a new contact using the information from a received email.
- Syncing the phone's contact list with the email server's contact list.

At this stage, instead of capturing every detail of the potential flow, we can instead assign each local resource a relative weight, based on its sensitivity and context. For example, we believe that the user's location is generally more sensitive than his phone state; the location resource would thus have a higher value than the phone state resource. Transit resources, by their nature, do not require a weight because they only protect data passage to the outside world, instead of some sensitive internal state. Since any given flow can leak anything, all edges are assumed to have infinite capacity, while the local resources have some predetermined capacity.

We can now measure the potential risk of this application by calculating the maximum flow between the user and world nodes. We call the maximum flow from the user to the world the *spy score* of the application. This score suggests the maximum value of the resources that the application can spy on, then leak out to the world. In a similar manner, we call the maximum flow from the world to the user the *bully score* of the application. This score suggests the maximum value of the resources that the outside world can tamper with on the user's device, through the use of the application.

[3] We observe that some orderings and flows may in fact be malicious, as suggested by Banuri *et al.* (2012).

We note that as a matter of convenience, instead of assigning each local resource node a value directly, all edges between the local resource and the user are instead set to the value of the node, while all other edges are implied to have infinite capacity.

To demonstrate how this works, consider the Candy Crush Soda app, which requests the following system permissions[4]

- com.android.vending.BILLING – Allows the app to make in-app purchases.
- INTERNET – Allows access to the internet and network sockets.
- ACCESS_NETWORK_STATE – Allows the app to check whether the device is connected to the internet (including cellular data).
- ACCESS_WIFI_STATE – Allows the app to access whether the device is connected to WiFi.
- GET_ACCOUNTS – Allows the app to request authorization tokens from the system.
- WAKE_LOCK – Allows the app to prevent the phone from sleeping.
- READ_CONTACTS – Allows the app to read from a list of available contacts.

With these permissions, Candy Crush has access to the user's accounts and contact information, resources that affect battery life (i.e. WAKE_LOCK), as well as the Internet, which is a transit resource. In other words, Candy Crush can read the all of the user's contacts and upload them to the world, as well as purchase content for the user.

[4]This app declares and requests a few additional custom permissions, which are not of concern here.

Resource	Weight
Account	10
Contacts	20
Network State	5
Battery	3

Table 4.1: This table shows the resource values for the Candy Crush Soda app. A full listing of these weights is elaborated on in Appendix A.

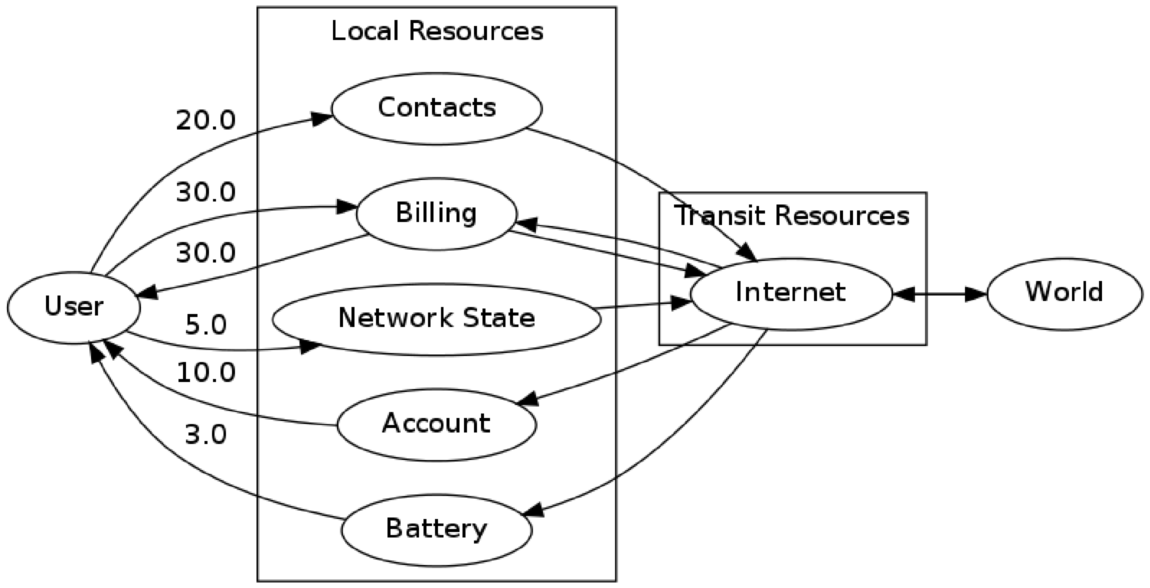


Figure 4.1: This shows the permission graph for applications with the same permissions as Candy Crush Soda. Note that edges without any edge weight are assumed to pass infinite flow.

If we assume that the user values his resources with the weights as assigned in Table 4.1, then the resulting graph is shown in Figure 4.1. Note that the weights of each resource and the implications of each permission for each resource are completely customizable; a more thorough explanation of the resource weights is given in Appendix A. The resulting spy score for Candy Crush Soda is 55, and the resulting bully score is 43. We note that Candy Crush may not leak everything to the internet, but these flows indicate the most aggressive actions that can be achieved with these permissions.

We make one other observation about this approach. Some apps are expected to inherently request more permissions than others. Facebook, for example, will request many different resources, such as access to contacts, location, calendar, and the camera, along with the internet. This is not particularly surprising, since it is a *social media* application. However, it would be strange for a flashlight application to request the same permissions; why would the flashlight need access to the user's location or contact information? Since the spy and bully scores are dependent only on the permissions requested, it only really makes sense to compare apps by category.

4.3 Application to Android Intents

Analyzing only the permissions of an application can be insufficient because applications do not always require access to a transit resource to leak data. As suggested in Section 3.3, Android applications can also send out *intents* to other applications installed on the user's device without declaring any permissions.

4.3.1 Code to Create Intents

For an app to send data to another app (or even to a different component of itself), an intent object must be created and sent to the system for resolution. This process used throughout Android and is required to even start the application from the home screen[5].

Before sending off an intent to the system for resolution, various aspects of the intent should be set. The code in Figure 4.2 illustrates a typical usage of creating an

[5] Any application that can be launched from the home screen must be able to receive an intent with a MAIN action and a LAUNCHER category [6].

```java
public void sendNoteText(String email_addr) {
  Intent intent = new Intent("android.intent.action.VIEW");
  intent.setType("text/plain");
  intent.putExtra("email", email_addr);
  startActivity(intent);
}
```

Figure 4.2: This shows a typical snippet of code to create an intent.

intent. The *sendNoteText* method will create an intent, then send it out to the world with the following properties:

Action android.intent.action.VIEW – A default action implying that the user wishes to *view* the file, whatever that means in this context.

Type text/plain – A MIME type which suggests how to interpret the data. In this case, it should be interpreted as a plaintext file.

Extra ⟨email,email_addr⟩ – A key-value pair of extras. The details of the extra fields are interpreted by the receiving application and often include sensitive meta-data, like the email address in this example.

The *startActivity()* call then instructs the Android system to resolve the intent to any Activity component that matches this intent.

4.3.2 Code to Receive Intents

Intents can be received only by one of the types of components listed in Table 3.1. For an application to receive intents, it defines an intent-filter in the AndroidManifest.xml that declares the properties of the intents that it can handle. For example, the intent created in Figure 4.2 can be resolved by a component with the intent-filter in Figure 4.3.

```
<activity class=".TextViewer">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/*" />
  </intent-filter>
</activity>
```

Figure 4.3: This shows a possible intent-filter for a component that can view and edit text.

4.3.3 Android Application Graphs

Examining the properties of intents allows for another use of the graph approach in Section 4.1 to model flows between applications. Since Android applications each have a state, and are initially isolated due to the Android sandbox, they can be modeled as nodes. We construct a directed edge between two nodes if the source node is capable of sending an intent to the sink. The properties of this edge are augmented with the properties of the intent.

For example, suppose that we have a notes application that can edit and export plain text, an addressbook app that can access and edit the user's contact information, a camera application that can take and share pictures, and an email application to send email. The Notes app may define an intent-filter like that in Figure 4.3, since it is capable of viewing and editing text data. The Notes app may also allow the user to export their notes, which would imply that it could send an intent with a *SEND* action and text data.

In a similar manner, the email app might define an intent-filter that allows for sending any type of data, like the one shown in Figure 4.4. Since the *SEND* action

```
<activity class=".ComposeEmailActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND" />
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="*/*" />
  </intent-filter>
</activity>
```

Figure 4.4: This shows a possible intent filter for an activity that will send Email.

from the Notes application can be handled by the Email app based on its intent-filter, we construct a directed edge from the Notes app to the Email app.

In an analogous manner, if the Addressbook app allows for the user to edit text data and send it out for various contacts and the camera app allows for sending images, then the resulting application graph can be represented by Figure 4.5. Whether the user wants these applications to all interact will depend on the task at hand.

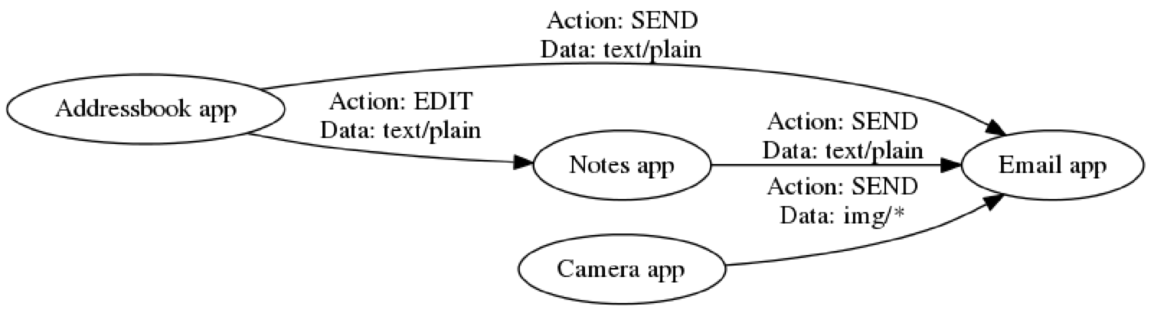


Figure 4.5: This shows a possible graph for the Notes app, Email app, Addressbook app, and Camera app as described in Section 4.3.3. The edges include the properties of the sent intent.

Chapter 5

IMPLEMENTATION

5.1 Analyzing Android Permissions

To calculate the flows for the requested resources of an Android application, we must first identify and determine the values of the differing resources. Most of the permissions protected resources that are self-explanatory, such as READ_CONTACTS. For the others, we read the documentation provided by Google Google (2015b).

We assigned the values for each resource based on our estimates of their sensitivity. Location information as well as Billing information were seen as the most sensitive, and so these resources were given a value of 30. Other resources, such as a user's contact information, voicemail, and SMS/MMS which were seen as sensitive were given a value of 20. Table 5.1 lists the values for some of the more common local resources that applications request; a more complete list is available in Appendix A.

To calculate the spy and bully scores of an application, we wrote the permission graph analyzer in Java. It constructs the graph based on the Resource Values file, which specifies the resources, their weights, along with the set of permissions that they respond to. The flows in the graph are calculated using the Edmond-Karp algorithm provided by the JGraphT library JGr (2015).

To acquire the permissions from an application, we wrote the permission graph analyzer to accept and parse the AndroidManifest.xml directly. However, since the AndroidManifest.xml file is not always directly available, it can also read the permissions from AAPT[1] output, or even from a newline-separated list. The process for

[1]Android Application Packaging Tool, which is part of the Android SDK

Local Resource	Weight
Contacts	20
External Storage	10
SMS/MMS	20
Location	30
Voicemail	20

Table 5.1: This stores the values of common resources that are used when calculating the spy and bully scores. This table only includes several common resources; a more complete list is in Appendix A.

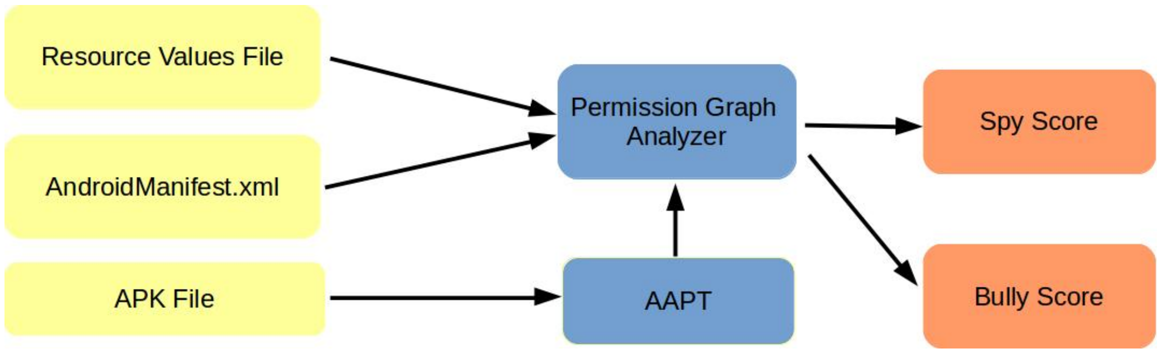


Figure 5.1: This shows the process for calculating the spy and bully scores for an application. Note that either the AndroidManifest.xml file can be used, or the output from the AAPT tool can be used if only the application APK is available.

calculating the scores is shown in Figure 5.1.

We also identified the categories from the Google Play store, which are fully listed in Appendix B. We use these implicit categories, provided by the Google Play store and set by the developer, to classify applications based on their functionality. While it is possible for a developer to miscategorize their application, they should not be inclined to do so; putting applications in the correct category increases exposure, since users should search a particular category if they already know what they are looking for.

We reiterate that even after considering each category, this approach can still only estimate the worst-case behavior of a given app based on its requested permissions. It is possible that the app does not leak these resources or uses them in a secure manner that the user would otherwise expect. This is an inherent limitation in assigning numeric edge weights; a single number cannot capture the entire nature of potential interactions between resources. This formulation is designed to quantify the possible risks of installing an application.

5.2 Analyzing Code

To construct the Application graphs for a collection of Android applications, we need to know the intents that an application can receive, as well as the intents it sends out. Extracting the intents that an application can receive can be done by parsing the AndroidManifest.xml file and searching for the available intent-filters; this is the precise purpose of intent-filters in the first place.

Extracting the intents that an application sends out, however, is more complicated. In many cases, the source code for Android applications is not directly available. Instead, we decompile the Android applications and search for specific method calls. We note that the calls to create an Intent instance (i.e. call its constructor) cannot be obfuscated easily since these calls are only available from the Android framework. Thus, malicious apps cannot obfuscate the creation of the Intent without dynamically loading code, or obfuscating the strings used to create it. Still, there are inherent limitations for decompilation and static analysis, which we address in Section 5.2.3.

5.2.1 Extracting Possible Sent Intents

In order to detect the types of intents that an app can send out, we trace the locations of where intents are created and follow them through the code as they are modified. The intent class in the Android framework has specific methods that modify

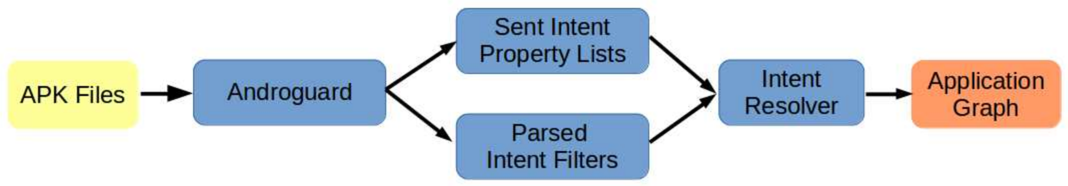


Figure 5.2: This shows the process of extracting the intents and intent-filters from an Android app. Since the source code for the application is not available, we decompile the APK files using Androguard and calculate the different intents that can be sent out and received. Finally, we determine which intents can be resolved by certain apps to construct the Application graph.

its internals; the full list of methods that we consider and what they do is listed in Appendix C.

Many of these modification functions take in strings as input arguments; these strings are very often constants that do not change at runtime. For example, *android.intent.action.VIEW* is a common action that is used by numerous intents throughout the Android system; this string is almost always a constant hard-coded string. Even some of the properties that do change (i.e. the *putExtra()* functions) will be indexed by a revealing key that does not, like the ”email” text in sample shown in Figure 4.2.

Our process for extracting the intents sent out by an application is as follows:

1. Identify locations in the code where intents are created.

2. Create a property list that holds the possible fields for the created intent. It is initially empty.

3. When an intent method is called, determine possible values for the input arguments. This can be done as follows:

```
new-instance v3, Intent;
const-string/jumbo v0, 'android.intent.action.SEND'
invoke-direct v3, v0, Intent;-><init>(String;)V
const-string/jumbo v0, 'plain/text'
invoke-virtual v3, v0, Intent;->setType(String;)Intent;
const-string/jumbo v4, 'android.intent.extra.SUBJECT'
invoke-direct v6, v0, getSubject;->h(I)String;
move-result-object v5
invoke-virtual v3, v4, v5, Intent;->putExtra(String; String;)Intent;
const-string/jumbo v4, 'android.intent.extra.TEXT'
```

Figure 5.3: This shows how part of the code in Figure 4.2 *might* compile into Dalvik instructions. Of course, the code could be compiled in different ways depending on the compiler and optimizations used. For brevity, the class and method names without their full namespace scoping; the *Intent* class in the above really has a fully scoped name of *android.content.Intent*, and the *String* class really has a fully scoped name of *java.lang.String*.

   (a) Backtrace to where the input arguments were set.

   (b) If the argument is a constant string or integer, add it to the possible values in the property list.

4. Return the property list, indicating all possible intents.

In order to extract these intents, we built the application graph analyzer with the open-source Androguard framework written by Desnos and Gueguen (2015). This framework is written in Python and decompiles Android applications directly from the APK file. The framework allows for printing the Dalvik instructions for each method, as well as identifying calls into the Android framework.

To explain the algorithm used to extract intents, consider code snippet in Figure 4.2. This code might compile into the Dalvik bytecode shown in Figure 5.3. It is clear that many of the constant strings are recognizable by the *const-string* class of instructions in Dalvik. By tracking which registers these constant strings are stored to and by tracking where these variables are used, it is possible to determine when Intent methods are called and what the values for some of their arguments are. This information, in turn, can be combined to generate a list of intents (and their properties) that the application can send out. In this analysis, other types of instructions, such as branch and jump instructions are ignored; this will be discussed in Section 5.2.3.

To perform the decompilation of applications, we used the open-source Androguard framework written by Desnos and Gueguen (2015). This framework is written in Python, and decompiles an application allowing for querying and tracing the bytecode for individual methods of an application.

5.2.2 Application Graph Generation Overview

Intents are resolved depending on the receiving application's intent filter as described in Section 3.3. Once the intents sent out by an application are known, they can be cross-referenced with another application's intent filter to see whether they communicate. Androguard was again used for this phase since it also allows for extracting and printing the intent filters from an application's AndroidManifest.xml file. After determining the intents sent out by an application, they are compared against every other application's intent filter to determine whether they can communicate. If they can, then an edge is added on the graph between these two applications with a weight that reflects the properties of the resolving intent(s).

An overview of this process to generate the application graphs is shown in Figure 5.2.

5.2.3 Limitations

The process of decompiling an application to determine its behavior has inherent limitations. Without executing the code itself, it is a difficult problem to determine what outputs a particular method will produce; this problem is computationally equivalent to the halting problem, which is undecidable in the general case. As such, we rely on constant strings and values to determine the properties of a sent intent, instead of executing the code directly. Our decompilation approach does not find any applications that obfuscate any strings used for intents or otherwise load them dynamically through some other means. We suggest that legitimate applications should not have any reason to hide the intents that they send out, and any applications that do obfuscate these strings are inherently suspicious[2].

Our treatment of branch and jump instructions is also not complete. Consider the example in Figure 5.4. Here, the two Extras parameters, *url* and *email* for the intent are mutually exclusive; however, the resulting list of sent intents will include an intent with both used simultaneously. While cases like these do arise, they are rare from our experience because most branch instructions used when creating intents are used to include additional data if available instead of mutually excluding two different parameters. It is usually to the developer's advantage to provide as much information as possible to another application resolving the intent for better usability. Furthermore, the application graph is trying to construct *possible* information flows, even if a particular branch is not all that likely in practice.

[2] It is possible that some applications (especially games) which use native code may create intents through the JNI interface instead of directly in Java. This is out of scope for this work, and we observe again that most applications considered here do not have native code to worry about.

```
//... Intent intent;
if(!hasEmail) {
  intent.putExtras("url", addr);
} else {
  intent.putExtras("email", addr);
}
//...
```

Figure 5.4: This code shows the limitations of branch instructions on our decompilation approach.

Chapter 6

RESULTS

To test these approaches, we downloaded the Top 20 free applications in the Google Play Store, as well as the top 10 applications from each category, except games and widgets[1]. In the cases of categories that did not offer a *Top Free* selection screen, we chose the first available selection. Some apps also were not compatible with the device type we registered the account for, in which case, we downloaded the next app on the list. Since these apps were chosen based on their ranking and accessibility on the Google Play store, we feel that the 10 selected apps are representative of their respective categories. Here, we analyze the results.

6.1 Permission Graph Analysis

We ran the Permission Graph Analyzer on the downloaded applications. The resulting scores for the Top 20 are displayed in the histogram in Figure 6.1, while the mean and standard deviation are shown in Table 6.1. We notice that in general,

Statistic	Value
Mean	73.4
Standard Deviation	41.27

Table 6.1: This shows the mean and standard deviation for the flow scores of the top 20 applications in the Google Play store.

[1] Games were not included because there were different subcategories to choose from, and because they were more likely to use native C/C++ code with JNI interface. Widgets were not included because many of the apps in this category were already downloaded and the category itself is rather vague. The full set of categories is listed in Appendix B.

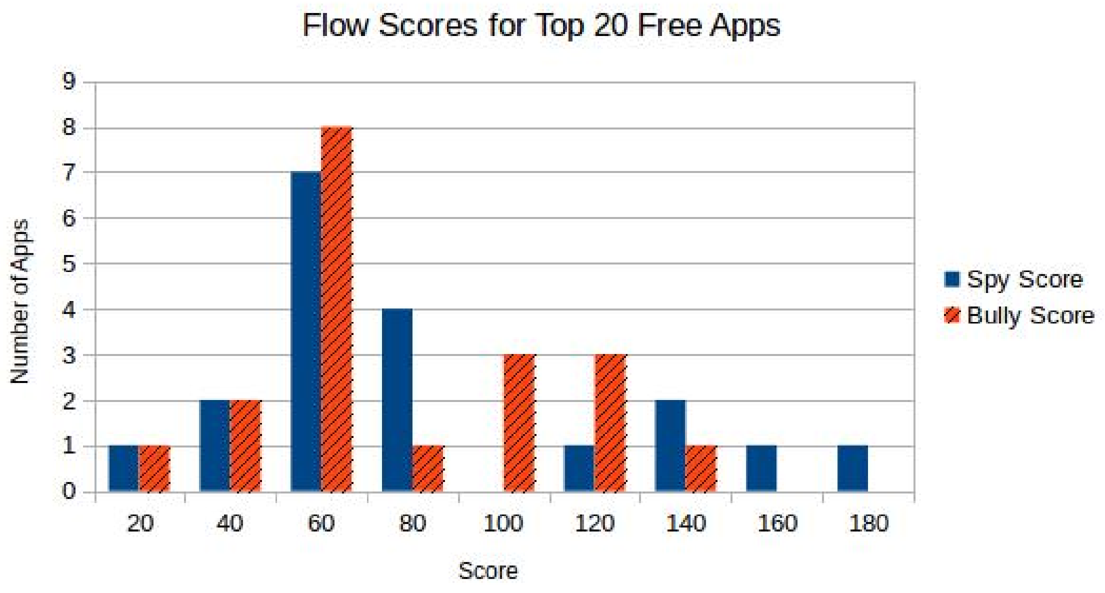


Figure 6.1: This chart shows a histogram of the flow scores for the top ten free applications from the Google Play Store. Observe how the spy scores are generally higher than the bully scores.

the spy scores are higher than the bully scores. This is likely due to the fact that reading and analyzing user information is usually required for writing, and is also generally more valuable for benign applications. We also notice from Table 6.1 that the standard deviation is rather high; the lowest score possible is 0, which falls less than two standard deviations below the mean. This suggests the inherent variability in applications on the Google Play store, due to the different scopes of different apps.

The mean flow scores were also calculated for each category and are displayed in Figure 6.2. These values suggest that certain categories of apps may require more resources than others. The mean scores for apps in the *Social* and *Communication* categories are both high, which is not surprising since apps in these apps often request access to the user's camera, microphone, location, and other such data. Apps in the *Personalization* category, on the other hand, seem to score much lower, probably because these apps only affect things like the user's dictionary, wallpaper, or other resources that are not as sensitive.

In addition to the mean score, we plot the distribution for the scores in each category using box plots, as shown in Figure 6.3 and Figure 6.4. These box plots also show outliers for each category, which are apps in the category whose score is 1.5 times the interquartile range.

The resulting means and distributions suggest commonalities between several categories. For example, the Communications category and the Social category both have similar means and distributions. This is due to the fact that many communication apps have social media features, like GroupMe or Yahoo Messenger.

One thing to note is that there are a fairly large number of outliers among all of the categories. Table 6.2 displays the list of applications that were outliers with spy scores *above* the median. We do not consider outliers below the median here because these apps that request less intrusive permissions than average are more desirable, at least in terms of risk.

Of the two outliers for the Education category, the Night Sky Tools application requests Camera and Location access, making the score higher. As per the description on Google Play, the application is an astronomy app that creates star maps for the user. It requests the Location and Camera permissions to create the appropriate star maps for the user, an action that is not common in the Education category as a whole. Given what the app does, we believe that it still falls into the Education category and its existence as an outlier is a rare case.

The other outlier in the Education category, TpT (Teachers pay Teachers), allows for teachers to share and sell content for their lesson plans. As such, the app allows for in-app purchases which boosts the spy and bully scores. This app appears to be more of a store application with an education theme than an actual Education app.

While some might still consider this as properly categorized, we still believe that it should receive extra care when downloading, since this app can make purchases on the user's behalf.

The two outliers in the Library category appear to be cases of miscategorization. The Cardboard application is advertised on the Google Play Store as an app that "puts virtual reality on your smartphone" Google (2015a). The app offers various functionality to view your videos and take virtual tours of various places around the world. This application really doesn't have much to do with a Library, and might be more appropriate in the Travel category, with more comparable scores. The INOVA Text to Speech app is designed to read text from different sources out loud. Again, this does not really fit into the Library category and might be more appropriately placed in either the Productivity, Tools, or Business categories, all of which have comparable scores.

Cymera is an interesting outlier because it is listed in the Photography app and advertises itself as Photo-editing app. Yet, Figure 6.3 shows that this application scores higher than 75% of all the top apps in the Social Media category. It requests access to the user's contacts, SMS/MMS, Location, as well as in-app purchases. This app is overzealous in what it requests and definitely deserves more scrutiny by the user before installing.

The other outliers are more straightforward. CM Security is an antivirus program, which inherently requires access to most of the system resources. Outside of an explicit Antivirus category, this really only fits into the Tools category. Waze is a GPS app with various social media plugs; it might be better categorized in the Social category. Talking Angela is a virtual pet app, which may be better classified as a game; however, the properties of the game category were not considered here.

One limitation that we acknowledge is that the downloaded apps are all free. Since all of these apps are free, there is a potential bias as free apps often have ads and other such mechanisms that can still profit the developer. Some developers even have a free and paid version of the same app, with ads disabled on the paid version.

6.2 Application Graph Analysis

For a simple example, we have generated the graph for interactions between the following apps in Figure 6.5:

- Skype
- Walmart
- Facebook

While it is not all that surprising that Facebook and Skype can intercommunicate, it is curious that the Walmart app can communicate with both. Walmart can most likely communicate with Skype to allow users to call a particular store, while it probably lets a user "like" Walmart on Facebook.

One application of interest alluded to earlier was the popular free *Super Bright LED Flashlight* app. This application has over 3 million downloads and numerous 5 star reviews. The Google Play store has classified it in the Productivity category. The spy score for this app is 35 and the bully score is 36; the permissions it requests score lower than average for the Productivity category.

However, when decompiling the application, we notice that this flashlight app has the ability to send out the intents shown in Table 6.3. We observe that many of these intents can be resolved directly by sensitive system applications, like the phone,

[2]Note that for space constraints, only the last portion of the full text of the action is listed here. For example, VIEW is really *android.intent.action.VIEW*. The borrowed convention is similar to that for Android permissions.

email, SMS, and the calendar. Figure 6.6 shows possible interactions with some of the Google apps, like Gmail and Hangouts.

6.3 Discussion

The distribution of the flow scores shown in Figure 6.3 and Figure 6.4 show a number of outliers, suggesting that these scorings can identify overzealous or miscategorized apps in certain categories. We observe that some categories had a much more diverse set of scores to consider, making them less effective in identifying outliers; the interquartile ranges for the *Business*, *Lifestyle*, *Productivity*, and *Transportation* all exceeded a value of 60. These categories are generally broader in terms of what they do; the Productivity category includes both the *Kingston Office* app with a spy score of 5 and the *Evernote* app which scores around 157. This suggests that a possible improvement might be to score different resources differently based on category.

One use of Permission Graphs and the resulting flow scores might be to augment the appearance and behavior of the Google Play store. The values and meanings of this score could be displayed whenever the user downloads an app; work by Felt et. al. has suggested that users can be responsive to some warning messages Felt *et al.* (2012).

Of course, some users ignore the existing warnings as it is, and are likely to ignore these new ones as well. In these cases, the scores could be used by the Google Play store more explicitly. When a developer classifies their app in one of the Google Play categories, the Google Play store could calculate the flow scores. Then, if that app is an outlier in its respective category, the maintainers of Google Play can identify this and take appropriate action, such as requiring the developer to reclassify their app, or requiring additional justification for the requested permissions.

Application graphs could also further augment the scoring. Since it can communicate with GMail, it is possible for the Flashlight app to access resources like SMS, email, and the phone, even though it never requests the desired permissions. In this way, the Flashlight app can be seen as implicitly having some of the permissions that GMail has access to, even though they are never explicitly requested. This case is particularly interesting because these types of transactions might not even involve a leakage of permissions that tools such as ComDroid will find. For example, on the system phone, the *DIAL* action requires no permissions and will load the number to be dialed, but will stop short of actually making the call, while the *CALL* action requires the proper permissions and can invoke the call directly. This difference is subtle enough that the user is not likely to notice.

Application graphs also show that simply analyzing Android permissions is not sufficient to understand the security of an app, even legitimate ones. Some apps can score low in terms of permissions, but still interact with sensitive resources via intents, as demonstrated by the Flashlight app. These graphs can also be used to generate additional information for the user before they download it; Google Play could use the application graphs to identify whether an app is capable of communicating with certain system apps (SMS, Phone, Email) and warn the user accordingly, in addition to the existing permissions.

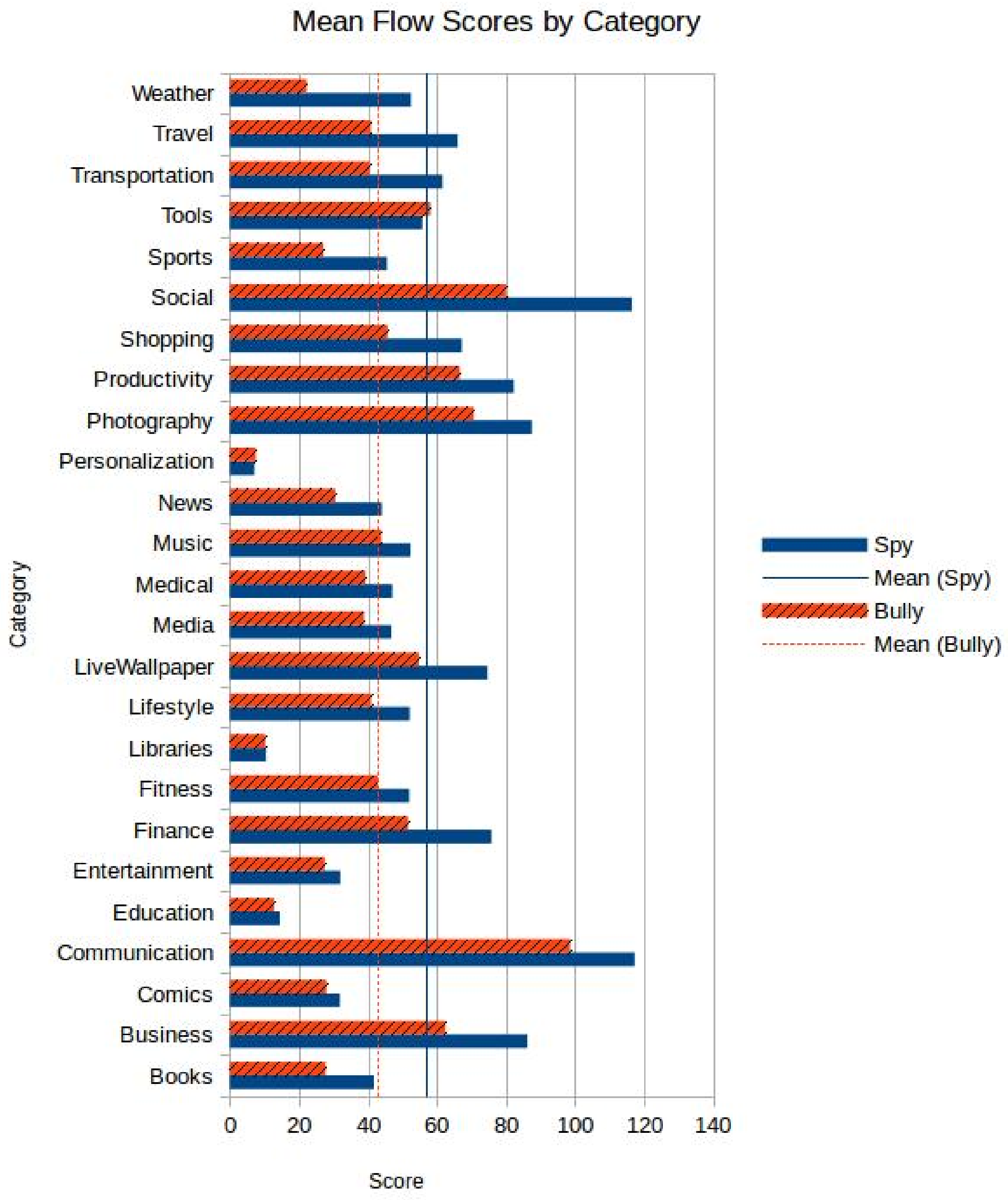


Figure 6.2: This chart shows the mean flow scores for each category of apps considered. The lines on the plot show the mean of all apps taken, regardless of category.

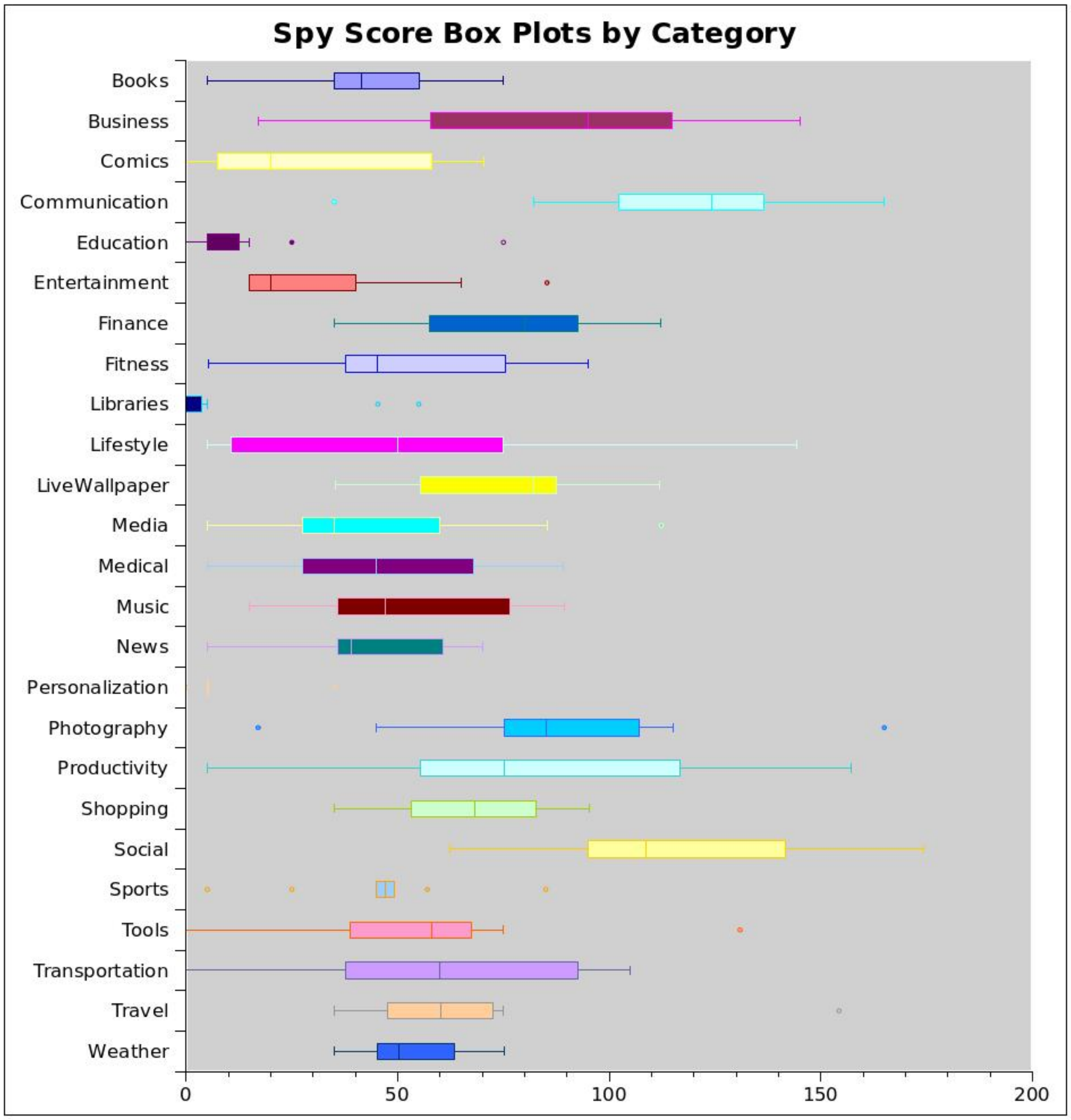


Figure 6.3: This shows the box-and-whisker plot for the spy scores by category. Notice the number of outliers, suggesting that some apps are miscategorized or promiscuous.

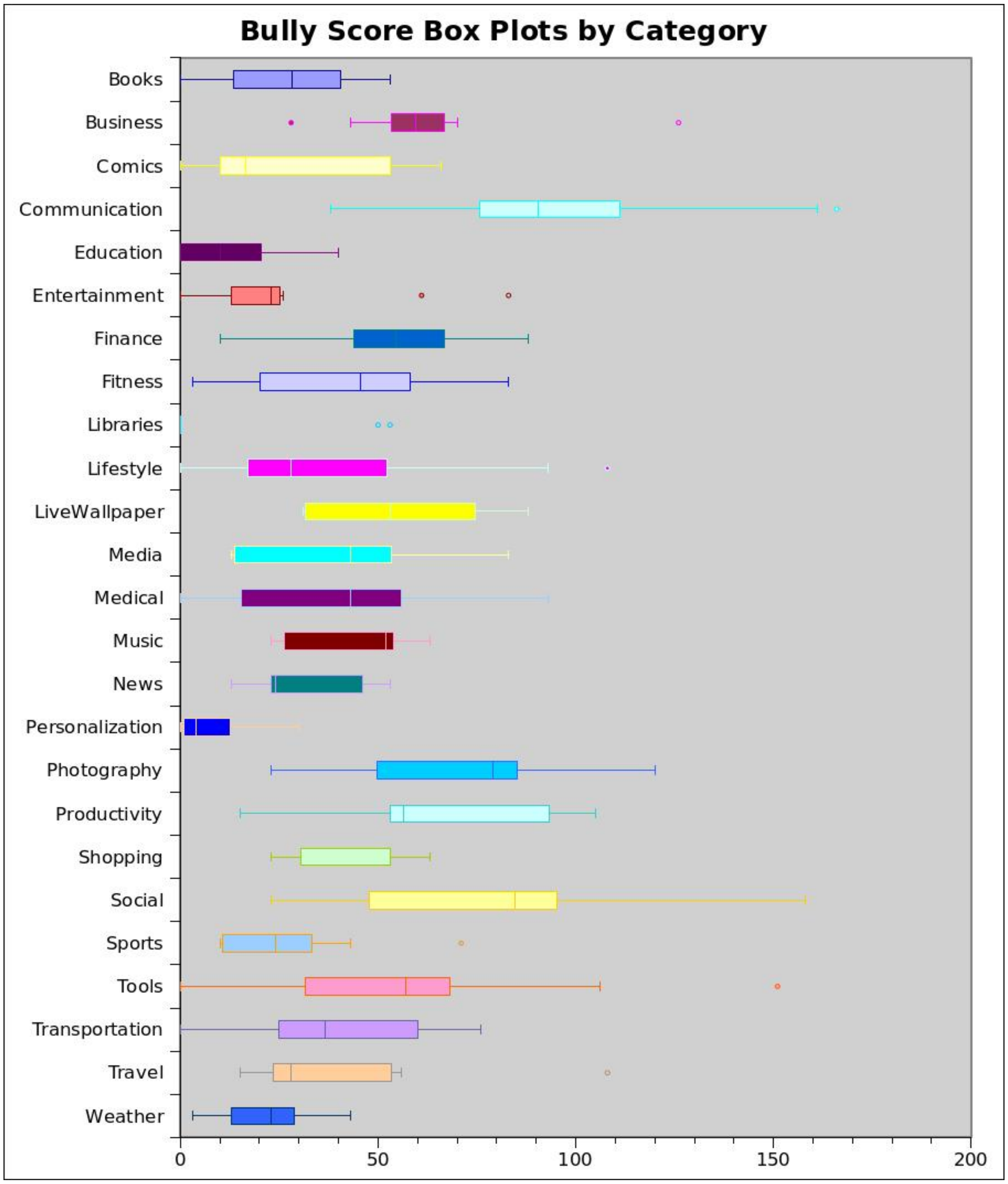


Figure 6.4: This shows the box-and-whisker plot for the bully scores by category.

App Name	Description
Education	
Night Sky Tools	This astronomy app uses location and the camera for custom star maps.
TpT	This app allows users to purchase teaching resources.
Entertainment	
Talking Angela	Virtual Pet application.
Libraries	
Cardboard	A Virtual Tour app
INOVA Text to Speech	App to read text out loud. Text can be read from other sources, like Google Maps.
Media	
Video Kik	Video-editing app
Photography	
Cymera	Photo-editing app
Tools	
CM Security	Antivirus app
Travel	
Waze	GPS App with Social Media features

Table 6.2: This table shows the applications that are outliers in their respective categories for the spy score. Note that the *Sports* category was omitted because the interquartile range was smaller than expected.

Method	Action[2]	Data	Extras
composeSms()	VIEW		sms_body
createCalendarEvent()	INSERT	vnd.android.cursor.item/event	
composeEmail()	SEND	plain/text	Email, Subject, Text
call()	CALL/VIEW		

Table 6.3: This table shows some of the different intents that the Super Bright LED Flashlight has the ability to send out. Notice that this flashlight application has the ability to send intents to apps that can send text messages, create calendar events on its behalf, even though it never requests the permissions necessary to do that on its own.

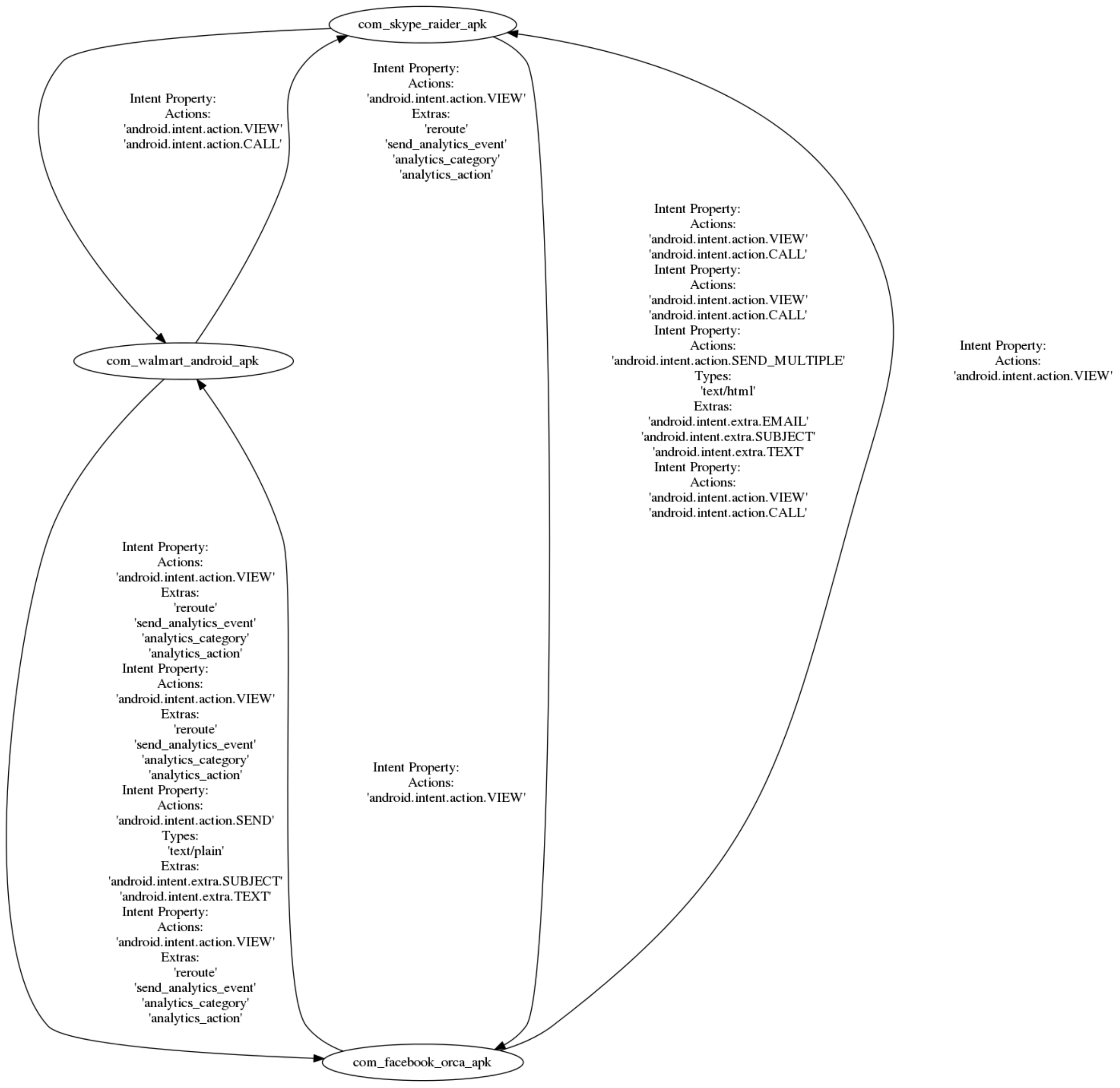
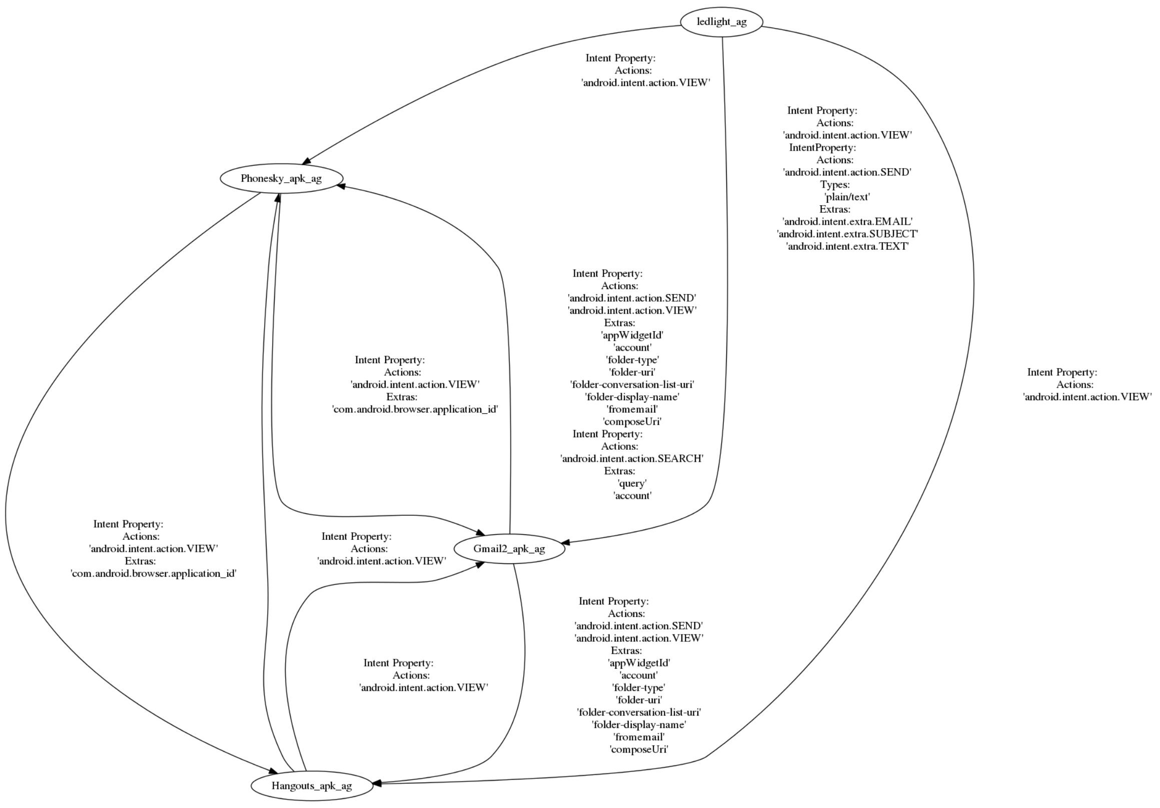

com_skype_raider_apk
com_walmart_android_apk
com_facebook_orca_apk
Intent Property: Actions: 'android.intent.action.VIEW' 'android.intent.action.CALL'
Intent Property: Actions: 'android.intent.action.VIEW' Extras: 'reroute' 'send_analytics_event' 'analytics_category' 'analytics_action'
Intent Property: Actions: 'android.intent.action.VIEW' 'android.intent.action.CALL' Intent Property: Actions: 'android.intent.action.VIEW' 'android.intent.action.CALL' Intent Property: Actions: 'android.intent.action.SEND_MULTIPLE' Types: 'text/html' Extras: 'android.intent.extra.EMAIL' 'android.intent.extra.SUBJECT' 'android.intent.extra.TEXT' Intent Property: Actions: 'android.intent.action.VIEW' 'android.intent.action.CALL'
Intent Property: Actions: 'android.intent.action.VIEW'
Intent Property: Actions: 'android.intent.action.VIEW' Extras: 'reroute' 'send_analytics_event' 'analytics_category' 'analytics_action' Intent Property: Actions: 'android.intent.action.VIEW' Extras: 'reroute' 'send_analytics_event' 'analytics_category' 'analytics_action' Intent Property: Actions: 'android.intent.action.SEND' Types: 'text/plain' Extras: 'android.intent.extra.SUBJECT' 'android.intent.extra.TEXT' Intent Property: Actions: 'android.intent.action.VIEW' Extras: 'reroute' 'send_analytics_event' 'analytics_category' 'analytics_action'
Intent Property: Actions: 'android.intent.action.VIEW'

Figure 6.5: This shows some possible interactions between the Walmart app, Skype, and Facebook.



ledlight_ag
Phonesky_apk_ag
Gmail2_apk_ag
Hangouts_apk_ag
Intent Property: Actions: 'android.intent.action.VIEW'
Intent Property: Actions: 'android.intent.action.VIEW' IntentProperty: Actions: 'android.intent.action.SEND' Types: 'plain/text' Extras: 'android.intent.extra.EMAIL' 'android.intent.extra.SUBJECT' 'android.intent.extra.TEXT'
Intent Property: Actions: 'android.intent.action.SEND' 'android.intent.action.VIEW' Extras: 'appWidgetId' 'account' 'folder-type' 'folder-uri' 'folder-conversation-list-uri' 'folder-display-name' 'fromemail' 'composeUri' Intent Property: Actions: 'android.intent.action.SEARCH' Extras: 'query' 'account'
Intent Property: Actions: 'android.intent.action.VIEW' Extras: 'com.android.browser.application_id'
Intent Property: Actions: 'android.intent.action.VIEW'
Intent Property: Actions: 'android.intent.action.VIEW' Extras: 'com.android.browser.application_id'
Intent Property: Actions: 'android.intent.action.VIEW'
Intent Property: Actions: 'android.intent.action.VIEW'
Intent Property: Actions: 'android.intent.action.SEND' 'android.intent.action.VIEW' Extras: 'appWidgetId' 'account' 'folder-type' 'folder-uri' 'folder-conversation-list-uri' 'folder-display-name' 'fromemail' 'composeUri'

Figure 6.6: This figure shows part of the graph between the popular Super Bright LED Flashlight app and some of the default Google apps.

Chapter 7

CONCLUSIONS AND FUTURE WORK

The security mechanisms provided by Android are incomplete. Understanding Android permissions require knowledge and patience that many users do not have. Even if they did, these permissions do not fully secure access to these resources since applications can still communicate using intents. Furthermore, permissions do not always track flows of information from the apps, which are often at odds with the user's expectation of what they have access to.

7.1 Summary

This work proposes a new way to look at Android security by examining information flow between apps using a graph-based approach. We first present a method for scoring an application based on the nature of its requested permissions. The resulting scores for each app were not effective on the apps at large, but were more effective when considering applications within their specific categories. This is because certain categories of applications, such as social media, inherently require more permissions. Outliers and extremities within each category show apps that are miscategorized or promiscuous.

We also present a method for analyzing the intercommunication between different applications, revealing some unexpected possible flows. We have identified some applications that do not request many permissions, yet they can still send out more than desired. The work here can aid the future design of a system to educate users on apps that can compromise their privacy.

7.2 Future Work

This work proposes a new way to look at Android security and how applications interact with each other using a graph-based approach. However, this work only focuses on constructing these graphs and analyzing their properties. One way this work could be improved is using the application graphs to modify the permission scoring; as it stands now, it is nontrivial to do this since the score will depend on the selection of apps in the application graph. The decompiler tool built to extract sent intents could also be improved to produce a more thorough listing of intents sent out by an application. Additional work could also work on capturing the sent intents at runtime and enforcing policies thereof. Finally, future studies could focus on user studies testing whether they can understand and customize security policies within a graph-based framework.

REFERENCES

"Jgrapht", `http://jgrapht.org/`, licensed under LGPL and EPL (2015).

Banuri, H., M. Alam, S. Khan, J. Manzoor, B. Ali, Y. Khan, M. Yaseen, M. N. Tahir, T. Ali, Q. Alam and X. Zhang, "An Android runtime security policy enforcement framework", Personal and Ubiquitous Computing pp. 1–11 (2012).

Chin, E., A. P. Felt, K. Greenwood and D. Wagner, "Analyzing inter-application communication in android", in "Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services", MobiSys '11, pp. 239–252 (ACM, New York, NY, USA, 2011), URL `http://doi.acm.org/10.1145/1999995.2000018`.

Desnos, A. and G. Gueguen, "Androguard", `https://github.com/androguard/androguard` (2015).

Enck, W., 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 "Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation", OSDI'10, pp. 1–6 (USENIX Association, Berkeley, CA, USA, 2010), URL `http://dl.acm.org/citation.cfm?id=1924943.1924971`.

Enck, W., P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, "Realtime privacy monitoring on smartphones", `http://appanalysis.org/demo/index.html` (2014).

Felt, A. P., E. Ha, S. Egelman, A. Haney, E. Chin and D. Wagner, "Android permissions: User attention, comprehension, and behavior", Tech. Rep. UCB/EECS-2012-26, EECS Department, University of California, Berkeley, URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-26.html` (2012).

Google, "Google play store", `http://play.google.com` (2015a).

Google, "Security tips", `http://source.android.com/devices/tech/security/` (2015b).

Holavanalli, S., D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Ko and L. Ziarek, "Flow permissions for android", in "Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on", pp. 652–657 (2013).

Ongtang, M., S. McLaughlin, W. Enck and P. McDaniel, "Semantically rich application-centric security in android", in "Computer Security Applications Conference, 2009. ACSAC '09. Annual", pp. 340–349 (2009).

Qiu, Y., "Bypassing android permissions: What you need to know", `http://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-what-you-need-to-know/`, [Online; accessed 15-April-2015] (2012).

Sbirlea, D., M. Burke, S. Guarnieri, M. Pistoia and V. Sarkar, "Automatic detection of inter-application permission leaks in android applications", IBM Journal of Research and Development **57**, 6, 10:1–10:12 (2013).

Surpax Technology Inc., "Super-bright led flashlight", `https://play.google.com/store/apps/details?id=com.surpax.ledflashlight.panel` (2015).

Tiwari, M., P. Mohan, A. Osheroff, H. Alkaff, E. Shi, E. Love, D. Song and K. Asanović, "Context-centric security", in "Presented as part of the 7th USENIX Workshop on Hot Topics in Security", (USENIX, Berkeley, CA, 2012), URL `https://www.usenix.org/conference/hotsec12/workshop-program/presentation/Tiwari`.

Xu, R., H. Saïdi and R. Anderson, "Aurasium: Practical policy enforcement for android applications", in "Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)", pp. 539–552 (USENIX, Bellevue, WA, 2012), URL `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu_rubin`.

Xu, W., X. Zhang and G.-J. Ahn, "Towards system integrity protection with graph-based policy analysis", in "Data and Applications Security XXIII", edited by E. Gudes and J. Vaidya, vol. 5645 of *Lecture Notes in Computer Science*, pp. 65–80 (Springer Berlin Heidelberg, 2009), URL `http://dx.doi.org/10.1007/978-3-642-03007-9_5`.

APPENDIX A

PERMISSION RESOURCE WEIGHTS

Here, we outline the full list of system permissions and how they map to certain resources. The weights of each resource and the permissions that pertain to that resource (including the direction of interaction) are fully configurable. This allows for adding resources as new permissions and features are added, as well as updating existing resources with different weights.

The list of initial resources that we considered are listed in Table A.1. The values and responding permissions of these resources are listed in Table A.2, though some have been omitted for brevity.

One problem that pervaded the assignment of the weights was the comparison of relative values between apps that requested many minor resources like Battery, Phone state, and so forth, and apps that only requested a few major resources like Location. As a result, the value of sensitive resources like Location is much higher than other non-sensitive resources, so that apps requesting many minor resources are not unfairly punished.

The general Permission Graph approach does not depend explicitly on the actual values of the resources[1], meaning that different values for different conditions are possible. In fact, the Permission Graph Analyzer tool itself assigns values based on the contents of a text file, which can be edited with different values as needed. Identifying better values of these resources or defining conditional values that depend on the particular category of the app under consideration is left as future work.

[1]Technically, the maximum flow algorithm approach does require that each Resource's value is non-negative.

[2]Note that for an app to access to any account, the user will be prompted whether to allow it.

Resource	Description
Contacts	The user's contacts and address book.
Storage	Access to external SD cards or other mediums.
Account	Access to authentication tokens for different accounts, like Facebook or Gmail.[2]
System	Access to system events, like boot, as well as some other informative fields.
Wallpaper	The user's selected wallpapers.
Camera	Access to the camera.
Battery	Features that may drain the battery more quickly.
SMS/MMS	Access to a user's text messages.
Device Alarm	Access to the device alarm.
Phone	Allows the user to make calls.
Bluetooth	Allows for using and setting up Bluetooth devices.
Sync Settings	Allows for querying whether apps can sync offline.
System Clock	Allows for updating the clock.
Location	Access to the user's location, using GPS and WiFi.
Browser Data	Access to browsing bookmarks and history.
System Display	Access to modal system dialogs.
Calendar	Access to the calendar.
Personal Profile	Access to the user's profile (contact) information on the device.
User Dictionary	Access to the user's spell-checking dictionary.
Microphone	Allows for recording audio.
Internet	Access to network sockets and the internet.
NFC	Near-field communication on some devices.
Voicemail	Access to Voicemail.
Billing	Allows for in-app purchases.
Network State	Connectivity information.
DRM	Access to new or custom DRM features.

Table A.1: This table lists the different resources and brief descriptions.

Name	Value	Type	Responding Permissions
Contacts	20	Local	READ/WRITE_CONTACTS
Storage	10	Local	READ/WRITE_EXTERNAL_STORAGE
Account	10	Local	GET_ACCOUNTS, AUTHENTICATE_ACCOUNTS, MANAGE_ACCOUNTS, USE_CREDENTIALS
System	3	Local	RECEIVE_BOOT_COMPLETED, CLEAR_APP_CACHE, GET_PACKAGE_SIZE, BROADCAST_STICKY
Wallpaper	2	Local	SET_WALLPAPER_HINTS, SET_WALLPAPER
Camera	30	Local	CAMERA
Battery	3	Local	TRANSMIT_IR, VIBRATE, WAKE_LOCK, FLASHLIGHT
SMS/MMS	20	Local	READ/WRITE_SMS, SEND/RECEIVE_SMS, RECEIVE_MMS
Device Alarm	5	Local	SET_ALARM
Phone	10	Local	CALL_PHONE, READ_PHONE_STATE
Bluetooth	N/A	Transit	BLUETOOTH, BLUETOOTH_ADMIN
Sync Settings	2	Local	READ_SYNC_STATS, READ-/WRITE_SYNC_SETTINGS, WRITE_SYNC_SETTINGS
System Clock	1	Local	SET_TIME_ZONE
Location	30	Local	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
Browser Data	5	Local	ACCESS_DOWNLOAD_MANAGER, READ-/WRITE_HISTORY_BOOKMARKS
System Display	5	Local	SYSTEM_ALERT_WINDOW
Calendar	10	Local	READ/WRITE_CALENDAR
Personal Profile	7	Local	READ/WRITE_PROFILE
User Dictionary	3	Local	READ/WRITE_USER_DICTIONARY
Microphone	10	Local	RECORD_AUDIO
Internet	N/A	Transit	INTERNET
NFC	N/A	Transit	NFC
Voicemail	20	Local	ADD_VOICEMAIL
Billing	30	Local	BILLING
Network State	5	Local	ACCESS/CHANGE_WIFI_STATE, ACCESS/CHANGE_NETWORK_STATE
DRM	30	Local	INSTALL_DRM, ACCESS_DRM

Table A.2: This shows the values and responding permissions for the system resources.

APPENDIX B

CATEGORIES IN THE GOOGLE PLAY STORE

At the time of this writing, the categories in the Google Play store that we considered are:

- Books & Reference
- Business
- Comics
- Communication
- Education
- Entertainment
- Finance
- Health & Fitness
- Libraries & Demo
- Lifestyle
- Live Wallpaper
- Media & Video
- Medical
- Music & Audio
- News & Audio
- Personalization
- Photography
- Productivity
- Shopping
- Social
- Sports
- Tools
- Transportation
- Travel & Local
- Weather

We did *not* consider these categories:

- Widgets
- Games – which has the following subcategories:
  - Action
  - Adventure
  - Arcade
  - Board
  - Card
  - Casino
  - Casual
  - Educational
  - Family
  - Music
  - Puzzle
  - Racing
  - Role Playing
  - Simulation
  - Sports
  - Strategy
  - Trivia
  - Word

APPENDIX C

INTENT METHODS

Here, we list the methods of the *android.content.Intent* class that we care about and their properties, derived directly from Google (2015b):

- addCategory() – Adds a new category to the intent.
- addFlags() – Adds flags to the intent.
- getAction() – Returns the action for this intent.
- get*Extra() – A series of methods that allow for getting the extras stored in this intent. The various names and overloads of this method allow for extracting specific Java types, like Char, Boolean, etc.
- getDataString() – Returns the data this intent is operating on.
- getExtras() – Returns a map of all the extras for this intent.
- getFlags() – Gets the flags for this intent.
- getPackage() – Returns the application package this intent is limited to.
- getType() – Returns any explicit MIME type included in the intent.
- put*Extra() – A series of method that allow for setting the extras stored in this intent.
- setAction() – Set the action for this intent.
- setDataAndType() – Set the data for this intent.
- setPackage() – Set an explicit application package name that limits the components this Intent will resolve to.
- setType() – Set an explicit MIME data type for this intent.

There are more methods than what is listed here. When decompiling, we check various regexes for the method names, since the input arguments for each of the methods is straightforward.