## Helping Developers Construct Secure Mobile Applications



Erika Chin

## Electrical Engineering and Computer Sciences University of California at Berkeley

Technical Report No. UCB/EECS-2013-58 http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-58.html

May 13, 2013

Copyright © 2013, by the author(s). All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

## Helping Developers Construct Secure Mobile Applications

By

Erika Michelle Chin

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy

 $\mathrm{in}$ 

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Wagner, Chair Professor Dawn Song Professor Brian Carver

Spring 2013

## Helping Developers Construct Secure Mobile Applications

Copyright © 2013 by Erika Michelle Chin

#### Abstract

### Helping Developers Construct Secure Mobile Applications

by

Erika Michelle Chin Doctor of Philosophy in Computer Science University of California, Berkeley Professor David Wagner, Chair

Mobile phones are no longer static devices that simply make phone calls and send SMS messages. Modern smartphones are now closer to general purpose computers. They allow users to customize their phones by installing third-party applications that let them browse the web, check social networking sites, and do online banking. Platform manufacturers, such as Android, introduce new APIs to facilitate the creation of rich applications that interact with other applications, system resources, and external resources (such as web applications). Given the level of trust users put in their phones and the number of sensitive tasks they perform, it is important to understand and improve the security of mobile applications.

Android provides tools to enable rich interaction, but if developers do not know how to use them correctly, they will not use them securely. In this dissertation, we examine how mobile applications interact with each other and their environment. We uncover threats to application security due to developer confusion and general misuse of the features provided by the mobile platform. Specifically, we perform an in-depth analysis of how Android applications interact with each other through inter-process communication mechanisms, how they interact with system resources through Android permissions, and how they interact with web content through WebViews. We build static analysis tools to identify vulnerable applications and measure the prevalence of the vulnerabilities. Through automated and manual analysis, we identify patterns that illustrate how developers misuse these features and make their application vulnerable to attack. We further provide platform-level, API-level, and design-level solutions to help developers and platform designers build secure applications and systems.

## Contents

#### i Contents Introduction 1 1 Analyzing Application Communication in Android 6 $\mathbf{2}$ 2.16 2.27 2.3Intent-Based Attack Surfaces 122.418 2.5222.630 2.736 Platform Evaluation 2.8Other Mobile Platforms 41 Related Work 2.9424344 3 **Analyzing Android Permissions** 453.1453.2The Android Permission System 46 3.3 503.4533.5Application Analysis Tool 563.6593.7Related Work 643.8653.9 66 Analyzing Web Interaction in Android 67 4 4.1 67 4.2Application and Web Interaction 68 4.370Attacks

|    | $\begin{array}{c} 4.4 \\ 4.5 \\ 4.6 \\ 4.7 \\ 4.8 \end{array}$ | Bifocals      | 73<br>78<br>83<br>86<br>86  |  |  |  |
|----|--|---------------|-----------------------------|--|--|--|
| 5  | Rela   | ated Work     | 88                          |  |  |  |
| 6  | Conclusion   |               |                             |  |  |  |
| A  | A.2  | Manual Review | <b>92</b><br>92<br>92<br>93 |  |  |  |
| Bi | bliog  | graphy        | 95                          |  |  |  |

#### Acknowledgments

I would like to thank the numerous people who have supported me over the years. First and foremost, I would like to thank my family. I am fortunate to have such loving parents whose unwavering confidence, steadfast support, and gentle encouragement have enabled me to pursue my interests and realize my potential. I thank my siblings who have given me a great deal of support in many forms: from my sister who is always just a phone call away when I want to chat, to my brother who is always willing to give me honest feedback. I thank my aunts, with whom I have shared many memorable holidays and vacations full joy and laughter (and full of delicious food).

I would like to express my utmost gratitude to my advisor, David Wagner. His mentorship and invaluable insights have shaped me into the researcher I am today. He taught me how to critically examine systems, discover problems worthy of research, and provide effective evaluations of proposed solutions. His enthusiasm and encouragement have helped me take a step back when I am being overly critical of my own work. By example, he reinforced the importance of taking the time to give back to the community.

I would also like to thank my quals and dissertation committee members, Eric Brewer, Brian Carver, and Dawn Song who so kindly gave their time and attention to give me valuable feedback on my work.

Finally, I would like to thank my colleagues and officemates, Adrienne Felt, Cynthia Sturton, Devdatta Akhawe, Matt Finifter, and Joel Weinberger. We shared many laughs both in the office and out, and I value their friendship. Whether working on the same research or not, their presence has made down-to-the-wire paper submissions much less stressful.

I give many thanks to the students, faculty, and staff at the UC Berkeley Computer Science Department, without which this dissertation would not be possible.

# Chapter 1 Introduction

Mobile phones have evolved from simple devices used for phone calls and SMS messages to sophisticated devices with full-fledged operating systems. Phone owners are no longer limited to the simple address book and other basic capabilities provided by the operating system and phone manufacturer. In fact, modern smartphones satisfy many of the same demands that desktop computers satisfy. Modern mobile phones provide wireless access, browser support, and the ability to customize the device by installing third-party applications. Users can check their email, update their social networking profiles, make bank transactions, and play mobile games, not only from their desktop computers, but also from their mobile phones. Mobile phones also provide unique capabilities not available on desktops. With access to GPS, users can get fine-grained, instant location updates. With access to the gyroscope, they now can have a portable portrait leveling tool.

It is not only the physical resources of smartphones that make them notably different from the traditional computer model. Modern mobile phones provide new paradigms and public APIs that allow developers to create applications with rich interactive functionality. These applications communicate with other applications, with protected resources provided by the phone, and with resources provided externally (e.g. across the network through web applications). For example, the Android platform provides a way for applications to expose their data and services to other third-party applications through Android's message passing service. These messages allow an application to invoke another application's service while giving the end-user the experience of using a single rich, multifunctional, seamless application. These public APIs introduce new paradigms that help developers create rich applications.

Another unique feature of smartphones is the centralized application repository (e.g., Android Market [3], BlackBerry App World [5], and iTunes App Store [9]). This has changed the application landscape for both users and developers. For developers, it has lowered the bar to entry for dabbling developers (through low, one-time developer fees) and provided a way for them to distribute their applications. For users, the application repository provides a straightforward, centralized way for users to browse and install applications. The combination of new design paradigms and the larger range in the skills of developers can lead to confusion and insecure code. We examine how mobile phone applications interact with each other and their environment and identify how using these rich features can lead to vulnerable applications. We help secure these applications by exposing common developer mistakes, providing static analysis tools to detect vulnerabilities, suggesting platform changes, and recommending developer education and better API documentation.

We examine three features of modern smartphone platforms.

#### Intents

First, we examine how applications interact with each other. Intents are messages in Android's inter- and intra-application message passing system. This application communication model promotes the development of rich applications. Android developers can leverage existing data and services provided by other applications while still giving the impression of a single, seamless application. For example, a restaurant review application can ask other applications to display the restaurant's website, provide a map with the restaurant's location, and call the restaurant. This communication model reduces developer burden and promotes functionality reuse.

Android's message passing system can become an attack surface if used incorrectly. The content of messages can be sniffed, modified, stolen, or replaced by malicious third-party applications, which can compromise user privacy. Also, a malicious application can inject forged or otherwise malicious messages, which can lead to breaches of user data and violate application security policies. We build a tool, ComDroid, that detects application communication vulnerabilities and find that 60% of applications analyzed have at least one vulnerability. We also find that developers commonly use the message passing system in a way that exposes both the message and the recipient to attack. We discuss insecure developer practices, the tool we built to identify exposed surfaces, and the vulnerabilities we found in the top 100 most popular applications in greater depth in the next chapter.

We further identify a subclass of communication vulnerabilities that arise due to developer confusion between inter-application and intra-application communication. We find that developers may publicly expose strictly internal messages. We find that 31% of the flaws we found with ComDroid are due to inter- vs. intra-application confusion. We further propose modifications to the Android platform to automatically detect and protect inter-application messages that should have been intra-application messages.

## Permissions

Second, we examine how applications interact with protected resources. Dangerous resources, such as resources that cost money (SMS messages or phone calls), resources that contain private data (access to the user's location or contacts), or device resources (bluetooth, WiFi), are protected by permissions. Developers must request these permissions upon application installation. As there is a disconnect between the permissions an application uses and the

permissions an application requests, it is possible to overprivilege an application (i.e., request more permissions than an application actually uses). This unnecessarily increases the capabilities of a successful attacker on an application. If an attacker can exploit a vulnerability in an application, it will receive access to any protected resources the application requested. We developed a framework for determining whether developers follow the principle of least privilege and request only the permissions that their application needs.

We analyzed 940 applications and found that 30.4% of applications are overprivileged. Our findings also show that most applications are overprivileged by only 1 permission, indicating that developers are trying to follow least privilege. Extraneous permission requests can be attributed to the lack of adequate documentation on the API permissions. Without this documentation, developers are left to guess which permissions are required. We have found that applications frequently have an extra permission that sounds similar to a permission they actually require. We have also found misleading advice on forums that say a permission is needed when it is not. The full details of our framework and analysis can be found in Chapter 3.

### WebViews

Third, we examine how applications interact with web content. Mobile devices and platforms are a rapidly expanding, divergent marketplace. Application developers are forced to contend with a multitude of Android mobile phones and tablets; customized OS branches (e.g., Kindle Fire, Nook Tablet); and a score of competing platforms including iOS and Windows Phone. Android developers are responding to the challenge of supporting multiple platforms through the use of WebViews, which allow HTML content to be displayed within an application. At a high level, WebViews provide the same functionality as a web browser, but allow full customizability with respect to how and what content is displayed (e.g., navigation UIs, full screen, etc). These in-application browsers allow developers to write code in platformneutral HTML and JavaScript that can be displayed by any Android device and version. Furthermore, application updates become simple. Developers merely update the HTML content downloaded by an application.

While convenient, these customized browsers can also pose a threat to application security, as allowing web content to interact with the application increases the application's attack surface. We show that these problems are real.

One unique feature of Android is that it provides a way for JavaScript in a WebView to invoke Android application code, if this is enabled by the application. In particular, the application developer can register an interface (an API to the mobile application) that can be called by the JavaScript. This allows the web page to access functionality and data exposed by the application. This may seem safe, as typically developers use WebViews to display trusted websites. However, it introduces a new risk [70]. If the user navigates the WebView to an untrusted malicious website, the malicious page may receive access to potentially sensitive application data. Similarly, if the application loads a page over HTTP and if the user is using an insecure WiFi network, a man-in-the-middle could inject malicious content into the page and mount a similar attack. Allowing JavaScript to invoke application code breaks traditional browser security models.

In Chapter 4, we detail various WebView-based attacks and present our vulnerability identification tool, Bifocals. We ran the tool on a data set of 864 applications. Among the 608 applications that contain WebViews, we find that over 20% of applications have the potential to give websites access to code. Of these applications, we find 54% allow a user to navigate to malicious JavaScript that could access application code.

Based on our findings, we recommend modifications to Android to address these risks. Our experiments suggest that these modifications would protect more than 60% of the vulnerable applications.

Although all of these features are provided by the platform with the intention of easing developer burden and promoting rich, powerful applications, this functionality can come at the cost of application security if not applied correctly. We show that when the API is not clear or not designed with security in mind, developers will create insecure applications. We recommend several approaches that platform designers and application developers can take to avoid these vulnerabilities.

First, platform behavior can be modified to close the vulnerabilities. This approach shifts the implementation burden from individual application developers to the platform developer. Instead of relying on developers to resolve their vulnerabilities (and waiting for them to make the fixes), platform changes could take effect with the push of one over-the-air update and be applied to currently installed applications. Some of the application features are enabled by default. For example, in the the WebViews chapter, we see that JavaScript is granted access to the filesystem by default. This is an unnecessary policy that creates the potential for attack. Features should be turned off by default, and developers should opt in when they want special functionality. Similarly, in the Intents chapter, we see that components are automatically made public when the developer includes an IntentFilter. This behavior is invisible and unintuitive. Developers should explicitly declare their access intentions.

Second, we propose extensions to the API to enforce security or improve the clarity of the functionality. Altering the platform behavior may not always be a reasonable approach, especially if we want to avoid breaking backward compatibility. Extensions to the API (e.g., adding methods) would apply secure practices to new applications without changing the behavior of existing methods. This approach would not affect legacy applications, which is both a benefit and drawback. While it avoids breaking legacy applications, it also does not secure them. It also has the drawback of being slower to take effect, relying on the pace at which new API versions are adopted. In the Intents chapter, we suggest extending the API to separate internal communication from external communication.

Third, we recommend developer education. Unfortunately, security is rarely a primary goal of application developers. By creating clear API documentation and discussing security issues with using the APIs, developers may use the APIs as intended and take steps to avoid the identified application vulnerabilities. In the permissions chapter, we recommend clear documentation on which APIs require which permissions in order to remove the guesswork by developers that contribute to overprivilege.

Fourth, we recommend that developers practice fundamental security principles such as principle of least privilege and separation of privilege. By separating privileged and unprivileged components, developers reduce the risk should a vulnerability be exploited.

Finally, we provide static analysis tools that developers can use to identify vulnerabilities in their applications. Through these five approaches, we believe we can improve the security of mobile applications and subsequently the security of the mobile device.

## Chapter 2

## Analyzing Application Communication in Android

## 2.1 Introduction

Android's application communication model is designed to help promote the development of rich applications. Android developers can leverage existing data and services provided by other applications while still giving the impression of a single, seamless application. For example, a restaurant review application can ask other applications to display the restaurant's website, provide a map with the restaurant's location, and call the restaurant. This communication model reduces developer burden and promotes functionality reuse. Android achieves this by dividing applications into components and providing a message passing system so that components can communicate within and across application boundaries.

Android's message passing system can become an attack surface if used incorrectly. In this chapter, we discuss the risks of Android message passing and identify insecure developer practices. If a message sender does not correctly specify the recipient, then an attacker could intercept the message and compromise its confidentiality or integrity. If a component does not restrict who may send it messages, then an attacker could inject malicious messages into it.

We have seen numerous malicious mobile phone applications in the wild. For example, SMS Message Spy Pro disguises itself as a tip calculator and forwards all sent and received SMS messages to a third party [87]; similarly, MobiStealth records SMS messages, call history, browser history, GPS location, and more [14,88]. This is worrisome because users rely on their phones to perform private and sensitive tasks like sending e-mail, taking pictures, and performing banking transactions. It is therefore important to help developers write secure applications that do not leak or alter user data in the presence of an adversary.

We examine the Android communication model and the security risks it creates, including personal data loss and corruption, phishing, and other unexpected behavior. We present ComDroid, a tool that analyzes Android applications to detect potential instances of these vulnerabilities. We used ComDroid to analyze 20 applications and found 34 vulnerabilities in 12 of the applications.

Most of these vulnerabilities stem from the fact that Intents can be used for both intraand inter-application communication, so we provide recommendations for changing Android to help developers distinguish between internal and external messages.

To automatically protect existing and future applications from these problems, we further propose a modification to the Android platform that heuristically infers unintended exposures and delivers those messages internally and prevents internal components from receiving external messages.

We find that 99.4% and 93.0% of applications are compatible with our sending and receiving changes, respectively. We find that our changes fix 100% of intra-application vulnerabilities found, which represents 31.4% of all security flaws identified by ComDroid. Our findings show that we can improve the security of applications with low backward-compatibility costs.

## 2.2 Android Overview

Android's security model differs significantly from the standard desktop security model. Android applications are treated as mutually distrusting principals; they are isolated from each other and do not have access to each others' private data. We provide an overview of the Android security model and inter-application communication facilities next.

Although other smartphone platforms have a similar security model, we focus on Android because it has the most sophisticated application communication system. The complexity of Android's message passing system implies it has the largest attack surface. In Section 2.8, we compare and contrast Android to other mobile operating systems.

## Threat Model

The Android Market contains a wide array of third-party applications, and a user may install applications with varying trust levels. Users install applications from unknown developers alongside trusted applications that handle private information such as financial data and personal photographs. For example, a user might install both a highly trusted banking application and a free game application. The game should not be able to obtain access to the user's bank account information.

Under the Android security model, all applications are treated as potentially malicious. Each application runs in its own process with a low-privilege user ID, and applications can only access their own files by default. These isolation mechanisms aim to protect applications with sensitive information from malware.

Despite their default isolation, applications can optionally communicate via message passing. Communication can become an attack vector. If a developer accidentally exposes functionality, then the application can be tricked into performing an undesirable action. If a developer sends data to the wrong recipient, then it might leak sensitive data. In this chapter, we consider how applications can prevent these kinds of communication-based attacks.

In addition to providing inter-application isolation, the Android security model protects the system API from malicious applications. By default, applications do not have the ability to interact with sensitive parts of the system API; however, the user can grant an application additional permissions during installation. We do not consider attacks on the operating system; instead, we focus on securing applications from each other.

### Intents

Android provides a sophisticated message passing system, in which *Intents* are used to link applications. An Intent is a message that declares a recipient and optionally includes data; an Intent can be thought of as a self-contained object that specifies a remote procedure to invoke and includes the associated arguments. Applications use Intents for both inter-application communication and intra-application communication. Additionally, the operating system sends Intents to applications as event notifications. Some of these event notifications are system-wide events that can only be sent by the operating system. We call these messages system broadcast Intents.

Intents can be used for *explicit* or *implicit* communication. An explicit Intent specifies that it should be delivered to a particular application specified by the Intent, whereas an implicit Intent requests delivery to any application that supports a desired operation. In other words, an explicit Intent identifies the intended recipient by name, whereas an implicit Intent leaves it up to the Android platform to determine which application(s) should receive the Intent. For example, consider an application that stores contact information. When the user clicks on a contact's street address, the contacts application needs to ask another application to display a map of that location. To achieve this, the contacts application could send an explicit Intent directly to Google Maps, or it could send an implicit Intent that would be delivered to any application that says it provides mapping functionality (e.g., Yahoo! Maps or Bing Maps). Using an explicit Intent guarantees that the Intent is delivered to the intended recipient, whereas implicit Intents allow for late runtime binding between different applications.

### Components

Intents are delivered to application *components*, which are logical application building blocks. Android defines four types of components:

- Activities provide user interfaces. Activities are started with Intents, and they can return data to their invoking components upon completion. All visible portions of applications are Activities.
- Services run in the background and do not interact with the user. Downloading a file or decompressing an archive are examples of operations that may take place in a

| To Receiver | <pre>sendBroadcast(Intent i)</pre>  |  |  |  |  |  |  |
|-------------|---|--|--|--|--|--|--|
|             | <pre>sendBroadcast(Intent i, String rcvrPermission)</pre>                           |  |  |  |  |  |  |
|             | <pre>sendOrderedBroadcast(Intent i, String rcvrPerm, BroadcastReceiver rcvr,)</pre> |  |  |  |  |  |  |
|             | <pre>sendOrderedBroadcast(Intent i, String rcvrPermission)</pre>                    |  |  |  |  |  |  |
|             | <pre>sendStickyBroadcast(Intent i)</pre>  |  |  |  |  |  |  |
|             | <pre>sendStickyOrderedBroadcast(Intent i, BroadcastReceiver receiver,)</pre>        |  |  |  |  |  |  |
| To Activity | To Activity startActivity(Intent i)   |  |  |  |  |  |  |
|             | <pre>startActivityForResult(Intent i, int requestCode)</pre>                        |  |  |  |  |  |  |
| To Service  | <pre>startService(Intent i)</pre>   |  |  |  |  |  |  |
|             | <pre>bindService(Intent i, ServiceConnection conn, int flags)</pre>                 |  |  |  |  |  |  |

Table 2.1: A non-exhaustive list of Intent-sending mechanisms

Service. Other components can *bind* to a Service, which lets the binder invoke methods that are declared in the target Service's interface. Intents are used to start and bind to Services.

- Broadcast Receivers receive Intents sent to multiple applications. Receivers are triggered by the receipt of an appropriate Intent and then run in the background to handle the event. Receivers are typically short-lived; they often relay messages to Activities or Services. There are three types of broadcast Intents: normal, sticky, and ordered. Normal broadcasts are sent to all registered Receivers at once, and then they disappear. Ordered broadcasts are delivered to one Receiver at a time; also, any Receiver in the delivery chain of an ordered broadcast can stop its propagation. Broadcasts Receivers have the ability to set their priority level for receiving ordered broadcasts. Sticky broadcasts remain accessible after they have been delivered and are re-broadcast to future Receivers.
- *Content Providers* are databases addressable by their application-defined URIs. They are used for both persistent internal data storage and as a mechanism for sharing information between applications.

Intents can be sent between three of the four components: Activities, Services, and Broadcast Receivers. Intents can be used to start Activities; start, stop, and bind Services; and broadcast information to Broadcast Receivers. (Table 2.1 shows relevant method signatures.) All of these forms of communication can be used with either explicit or implicit Intents. By default, a component receives only internal application Intents (and is therefore not externally invocable).

## **Component Declaration**

To receive Intents, a component must be declared in the application *manifest*. A manifest is a configuration file that accompanies the application during installation. A developer uses

9

the manifest to specify what external Intents (if any) should be delivered to the application's components.

#### **Exporting a Component**

For a Service or Activity to receive Intents, it must be declared in the manifest. (Broadcast Receivers can be declared in the manifest or at runtime.) A component is considered *exported*, or public, if its declaration sets the **EXPORTED** flag or includes at least one *Intent filter*. Exported components can receive Intents from other applications, and Intent filters specify what type of Intents should be delivered to an exported component.

Android determines which Intents should be delivered to an exported component by matching each Intent's fields to the component's declaration. An Intent can include a component name, an action, data, a category, extra data, or any subset thereof. A developer sends an explicit Intent by specifying a recipient component name; the Intent is then delivered to the component with that name. Implicit Intents lack component names, so Android uses the other fields to identify an appropriate recipient.

An Intent filter can constrain incoming Intents by action, data, and category; the operating system will match Intents against these constraints. An *action* specifies a general operation to be performed, the *data* field specifies the type of data to operate on, and the *category* gives additional information about the action to execute. For example, a component that edits images might define an Intent filter that states it can accept any Intent with an EDIT action and data whose MIME type is image/\*. For a component to be an eligible recipient of an Intent, it must have specified each action, category, and data contained in the Intent in its own Intent filter. A filter can specify more actions, data, and categories than the Intent, but it cannot have less.

Multiple applications can register components that handle the same type of Intent. This means that the operating system needs to decide which component should receive the Intent. Broadcast Receivers can specify a priority level (as an attribute of its Intent filter) to indicate to the operating system how well-suited the component is to handle an Intent. When ordered broadcasts are sent, the Intent filter with the highest priority level will receive the Intent first. Ties among Activities are resolved by asking the user to select the preferred application (if the user has not already set a default selection). Competition between Services is decided by randomly choosing a Service.

It is important to note that Intent filters are not a security mechanism. A sender can assign any action, type, or category that it wants to an Intent (with the exception of certain actions that only the system can send), or it can bypass the filter system entirely with an explicit Intent. Conversely, a component can claim to handle any action, type, or category, regardless of whether it is actually well-suited for the desired operation.

### Protection

Android restricts access to the system API with permissions, and applications must request the appropriate permissions in their manifests to gain access to protected API calls. Applications can also use permissions to protect themselves. An application can specify that a caller must have a certain permission by adding a permission requirement to a component's declaration in the manifest or setting a default permission requirement for the whole application. Also, the developer can add permission checks throughout the code. Conversely, a broadcast Intent sender can limit who can receive the Intent by requiring the recipient to have a permission. (This protection is only available to broadcast Intents and not available to Activity or Service Intents.) Applications can make use of existing Android permissions or define new permissions in their manifests.

All permissions have a protection level that determines how difficult the permission is to acquire. There are four protection levels:

- Normal permissions are granted automatically.
- *Dangerous* permissions can be granted by the user during installation. If the permission request is denied, then the application is not installed.
- *Signature* permissions are only granted if the requesting application is signed by the same developer that defined the permission. Signature permissions are useful for restricting component access to a small set of applications trusted and controlled by the developer.
- SignatureOrSystem permissions are granted if the application meets the Signature requirement or if the application is installed in the system applications folder. Applications from the Android Market cannot be installed into the system applications folder. System applications must be pre-installed by the device manufacturer or manually installed by an advanced user.

Applications seeking strong protection can require that callers hold permissions from the higher categories. For example, the BRICK permission can be used to disable a device. It is a Signature-level permission defined by the operating system, which means that it will only be granted to applications with the same signature as the operating system (i.e., applications signed with the phone manufacturer's signature). If a developer were to protect her component with the BRICK permission, then only an application with that permission (e.g., a Google-made application) could use that component. In contrast, a component protected with a Normal permission is essentially unprotected because any application can easily obtain the permission.

## 2.3 Intent-Based Attack Surfaces

We examine the security challenges of Android communication from the perspectives of Intent senders and Intent recipients. First, we discuss how sending an Intent to the wrong application can leak user information. Data can be stolen by eavesdroppers and permissions can be accidentally transferred between applications. Next, we consider vulnerabilities related to receiving *external* Intents, i.e., Intents coming from other applications. If a component is accidentally made public, then external applications can invoke its components in surprising ways or inject malicious data into it. Last, we summarize guidelines for secure Android communication.

Throughout our discussion of component security, we focus our attention on exported components. Non-exported components are not accessible to other applications and thus are not subject to the attacks we present here. We also exclude exported components and broadcast Intents that are protected with permissions that other applications cannot acquire. As explained in Section 2.2, Normal and Dangerous permissions do not offer components or Intents very strong protection: Normal permissions are granted automatically, and Dangerous permissions are granted with user approval. Signature and SignatureOrSystem permissions, however, are very difficult to obtain. We consider components and broadcast Intents that are protected with Signature or SignatureOrSystem permissions as private.

## Unauthorized Intent Receipt

When an application sends an implicit Intent, there is no guarantee that the Intent will be received by the intended recipient. A malicious application can intercept an implicit Intent simply by declaring an Intent filter with all of the actions, data, and categories listed in the Intent. The malicious application then gains access to all of the data in any matching Intent, unless the Intent is protected by a permission that the malicious application lacks. Interception can also lead to control-flow attacks like denial of service or phishing. We consider how attacks can be mounted on Intents intended for Broadcast Receivers, Activities, and Services. We also discuss special types of Intents that are particularly dangerous if intercepted.

#### **Broadcast Theft**

Broadcasts can be vulnerable to passive eavesdropping or active denial of service attacks (Figure 2.1). An eavesdropper can silently read the contents of a broadcast Intent without interrupting the broadcast. Eavesdropping is a risk whenever an application sends a public broadcast. (A public broadcast is an implicit Intent that is not protected by a Signature or SignatureOrSystem permission.) A malicious Broadcast Receiver could eavesdrop on all public broadcasts from all applications by creating an Intent filter that lists all possible actions, data, and categories. There is no indication to the sender or user that the broadcast has been read. Sticky broadcasts are particularly at risk for eavesdropping because they



Figure 2.1: Broadcast Eavesdropping (left): Expected recipients Bob and Carol receive the Intent, but so does Eve. Broadcast Denial of Service for Ordered Broadcasts (right): Eve steals the Intent and prevents Bob and Carol from receiving it.

persist and are re-broadcast to new Receivers; consequently, there is a large temporal window for a sticky broadcast Intent to be read. Additionally, sticky broadcasts cannot be protected by permissions.

Furthermore, an active attacker could launch denial of service or data injection attacks on ordered broadcasts. Ordered broadcasts are serially delivered to Receivers in order of priority, and each Receiver can stop it from propagating further. If a malicious Receiver were to make itself a preferred Receiver by registering itself as a high priority, it would receive the Intent first and could cancel the broadcast. Non-ordered broadcasts are not vulnerable to denial of service attacks because they are delivered simultaneously to all Receivers. Ordered broadcasts can also be subject to malicious data injection. As each Receiver processes the Intent, it can pass on a result to the next Receiver; after all Receivers process the Intent, the result is returned to the sending component. A malicious Receiver can change the result, potentially affecting the sender and all other receiving components.

When a developer broadcasts an Intent, he or she must consider whether the information being sent is sensitive. Explicit broadcast Intents should be used for internal application communication, to prevent eavesdropping or denial of service. There is no need to use implicit broadcasts for internal functionality. At the very least, the developer should consider applying appropriate permissions to Intents containing private data.

#### **Activity Hijacking**

In an *Activity hijacking* attack, a malicious Activity is launched in place of the intended Activity. The malicious Activity registers to receive another application's implicit Intents, and it is then started in place of the expected Activity (Figure 2.2).



Figure 2.2: Activity/Service Hijacking (left): Alice accidentally starts Eve's component instead of Bob's. False Response (right): Eve returns a malicious result to Alice. Alice thinks the result comes from Bob.

In the simplest form of this attack, the malicious Activity could read the data in the Intent and then immediately relay it to a legitimate Activity. In a more sophisticated active attack, the hijacker could spoof the expected Activity's user interface to steal user-supplied data (i.e., phishing). For example, consider a legitimate application that solicits donations. When a user clicks on a "Donate Here" button, the application uses an implicit Intent to start another Activity that prompts the user for payment information. If a malicious Activity hijacks the Intent, then the attacker could receive information supplied by the user (e.g., passwords and money). Phishing attacks can be mounted convincingly because the Android UI does not identify the currently running application. Similarly, a spoofed Activity can lie to the user about an action's completion (e.g., telling the user that an application was successfully uninstalled when it was not).

Activity hijacking is not always possible. When multiple Activities match the same Intent, the user will be prompted to choose which application the Intent should go to if a default choice has not already been set. (Figure 2.3 shows the dialog.) If the secure choice is obvious, then the attack will not succeed. However, an attacker can handle this challenge in two ways. First, an application can provide a confusing name for a component to fool the user into selecting the wrong application. Second, the malicious application can provide a useful service so that the user willingly makes it the default application to launch. For example, a user might opt to make a malicious browser the default browser and never get prompted to choose between components again. Although the visibility of the Activity chooser represents a challenge for the attacker, the consequences of a successful attack can be severe.

If an Activity hijacking attack is successful, the victim component may be open to a secondary *false response* attack. Some Activities are expected to return results upon completion. In these cases, an Activity hijacker can return a malicious response value to its invoker. If the victim application trusts the response, then false information is injected into the victim application.



Figure 2.3: The user is prompted when an implicit Intent resolves to multiple Activities.

## Service Hijacking

Service hijacking occurs when a malicious Service intercepts an Intent meant for a legitimate Service. The result is that the initiating application establishes a connection with a malicious Service instead of the one it wanted. The malicious Service can steal data and lie about completing requested actions. Unlike Activity hijacking, Service hijacking is not apparent to the user because no user interface is involved. When multiple Services can handle an Intent, Android selects one at random; the user is not prompted to select a Service.

As with Activity hijacking, Service hijacking can enable the attacker to spoof responses (a false response attack). Once the malicious Service is bound to the calling application, then the attacker can return arbitrary malicious data or simply return a successful result without taking the requested action. If the calling application provides the Service with callbacks, then the Service might be able to mount additional attacks using the callbacks.

## **Special Intents**

Intents can include URIs that reference data stored in an application's Content Provider. In case the Intent recipient does not have the privilege to access the URI, the Intent sender can set the FLAG\_GRANT\_READ\_URI\_PERMISSION or FLAG\_GRANT\_WRITE\_URI\_PERMISSION flags on the Intent. If the Provider has allowed URI permissions to be granted (in the manifest), this will give the Intent recipient the ability to read or write the data at the URI. If a malicious component intercepts the Intent (in the ways previously discussed), it can access the data



Figure 2.4: Intent Spoofing. A malicious application Mallory sends an Intent to an exported component Alice. Alice does not expect to receive a message from Mallory.

URI contained in the Intent. If the data is intended to be private, then an Intent carrying data privileges should be explicitly addressed to prevent interception.

Similarly, pending Intents delegate privileges. A *pending Intent* is made by one application and then passed to another application to use. The pending Intent retains all of the permissions and privileges of its creator. The recipient of a pending Intent can send the Intent to a third application, and the pending Intent will still carry the authority of its creator. If a malicious application obtains a pending Intent, then the authority of the Intent's creator can be abused.

## **Intent Spoofing**

A malicious application can launch an *Intent spoofing* attack by sending an Intent to an exported component that is not expecting Intents from that application (Figure 2.4). If the victim application takes some action upon receipt of such an Intent, the attack can trigger that action. For example, this attack may be possible when a component is exported even though it is not truly meant to be public. Although developers can limit component exposure by setting permission requirements in the manifest or dynamically checking the caller's identity, they do not always do so.

#### Malicious Broadcast Injection

If an exported Broadcast Receiver blindly trusts an incoming broadcast Intent, it may take inappropriate action or operate on malicious data from the broadcast Intent. Receivers often pass on commands and/or data to Services and Activities; if this is the case, the malicious Intent can propagate throughout the application.

Broadcast Receivers that register to receive Intents with system actions are particularly at risk of malicious broadcast injection. As discussed in Section 2.2, some Intents can only be broadcast by the operating system to inform applications about system events. These Intents contain action strings that only the operating system may add to broadcast Intents. (See Appendix A.3 for examples of system action strings). However, if a Broadcast Receiver registers to receive a system broadcast, the component becomes publicly accessible. In this case, a malicious application can send an Intent explicitly addressed to the target Receiver, without containing the system action string. If the Receiver does not check the Intent's action, then the Receiver will be tricked into performing functionality that only the system should be able to trigger.

### Malicious Activity Launch

Exported Activities can be launched by other applications with either explicit or implicit Intents. This attack is analogous to cross-site request forgeries (CSRF) on websites [28,60]. In most cases, a malicious Activity launch would just be an annoyance to the user because the target Activity's user interface would load. However, three types of Activity launching attacks are possible. First, launching an Activity can cause it to affect application state or modify data in the background. If the Activity uses data from the Intent without verifying the origin of the Intent, the application's data store could be corrupted. Second, a user can be tricked. For example, a user might click on a "Settings" screen in a malicious application, which directs the user to a screen in a victim application. The user might then make changes to the victim application while believing she is still interacting with the malicious application. Third, a victim Activity could leak sensitive information by returning a result to its caller upon completion.

#### Malicious Service Launch

If a Service is exported and not protected with strong permissions, then any application can start and bind to the Service. Depending on the duties of a particular Service, it may leak information or perform unauthorized tasks. Services sometimes maintain singleton application state, which could be corrupted.

A malicious Service launch is similar to a malicious Activity launch, but Services typically rely on input data more heavily than Activities. Consequently, a malicious launch attack where the Intent contains data is more likely to put a Service at risk. Additionally, there are more opportunities for a bound Service to return private data to its caller because Services often provide extensive interfaces that let their binders make many method calls.

## Secure Communication Guidelines

Developers should be cautious about sending implicit Intents and exporting components. When sending private data, applications should use explicit Intents if possible. Internal communication can and should always use explicit Intents. If it is not possible to use explicit Intents, then the developer should specify strong permissions to protect the Intent. Results returned by other components in response to Intents need to be verified to ensure that they are valid results from an expected source.

To make components more secure, developers should avoid exporting components unless the component is specifically designed to handle requests from other applications. Developers should be aware that declaring an Intent filter will export the component, exposing it to attack. Critical, state-changing actions should not be placed in exported components. If a single component must handle both inter- and intra-application requests, perhaps that component should be divided into separate components, one for each type. If a component must be exported (e.g., to receive system broadcasts), then the component should dynamically check the caller's identity prior to performing any operations. The return values of exported components can also leak private data, so developers should check the caller's identity prior to returning sensitive values. Intent filters are not security measures and can be bypassed with explicit Intents. Requiring Signature or SignatureOrSystem permissions is an effective way of limiting a component's exposure to a set of trusted applications. See Appendix A.2 for examples of how to create secure code.

## 2.4 ComDroid

We provide a tool, ComDroid, to detect potential vulnerabilities in Android applications. Applications for the Android platform include Dalvik executable (DEX) files that run on Android's Dalvik Virtual Machine. We first disassemble application DEX files using the publicly available Dedexer tool [79]. ComDroid parses the disassembled output from Dedexer and logs potential component and Intent vulnerabilities. We list the types of warnings ComDroid produces, separated by component and Intent type in Table 2.2.

## Permission Map

Every permission can be associated with one of the four protection levels described in Section 2.2. We consider both system-defined permissions (found in the system's Android Manifest [4]) and application-defined permissions (found in application manifests). We view Normal and Dangerous permissions as easy to obtain (or "weak") and consider components protected with those permissions as public.

## **Intent Analysis**

ComDroid examines Intent creation and transmission to detect the kinds of vulnerabilities outlined in Section 2.3. To do this, ComDroid statically analyzes disassembled output from Dedexer. Static analysis has commonly been used for bug finding [35, 68, 90]. ComDroid specifically performs flow-sensitive, intraprocedural static analysis, augmented with limited interprocedural analysis that follows method invocations to a depth of one method call. ComDroid parses translated Dalvik files and tracks the state of Intents, IntentFilters, registers, sinks (e.g., sendBroadcast(), startActivity(), etc.), and components. For each method that uses Intents (whether it is passed an Intent parameter, instantiates an Intent, or otherwise receives an Intent), ComDroid tracks the value of each constant, string, class, Intent, and IntentFilter. When an Intent object is instantiated, passed as a method parameter, or obtained as a return value, ComDroid tracks all changes to it from its source to its

| Unauthorized Intent Receipt  |                                       |  |  |  |  |  |  |
|------------------------------|---------------------------------------|--|--|--|--|--|--|
| Intent type                  | Potential vulnerability               |  |  |  |  |  |  |
| Sent Broadcasts              | Broadcast Theft (without data)        |  |  |  |  |  |  |
|                              | Broadcast Theft (with data)           |  |  |  |  |  |  |
| Sent Activity requests       | Activity Hijacking (without data)     |  |  |  |  |  |  |
|                              | Activity Hijacking (with data)        |  |  |  |  |  |  |
| Sent Service requests        | Service Hijacking (without data)      |  |  |  |  |  |  |
|                              | Service Hijacking (with data)         |  |  |  |  |  |  |
| Intent Spoofing              |                                       |  |  |  |  |  |  |
| Component type               | Potential vulnerability               |  |  |  |  |  |  |
| Exported Broadcast Receivers | Broadcast Injection (without data)    |  |  |  |  |  |  |
|                              | Broadcast Injection (with data)       |  |  |  |  |  |  |
|                              | System Broadcast without Action Check |  |  |  |  |  |  |
| Exported Activities          | Activity Launch (without data)        |  |  |  |  |  |  |
|                              | Activity Launch (with data)           |  |  |  |  |  |  |
| Exported Services            | Service Launch (without data)         |  |  |  |  |  |  |
|                              | Service Launch (with data)            |  |  |  |  |  |  |

Table 2.2: The list of different vulnerabilities associated with each type of Intent and component. "Without data" indicates the Intent involved in the attack does not contain extra data, whereas "with data" indicates the Intent does contain extra data in it and thus may additionally be vulnerable to data leakage or injection.

sink. An Intent sink is a call that transmits an Intent to another component, such as the calls listed in Table  $2.1.^1$ 

For each Intent object, we track (1) whether it has been made explicit, (2) whether it has an action, (3) whether it has any flags set, and (4) whether it has any extra data. For each sink, we check whether it is possible for any implicit Intent object to flow to that sink. Some Intent-sending mechanisms allow the sender to specify a permission that restricts who the Intent can be delivered to; our analysis records this information.

ComDroid issues a warning when it detects an implicit Intent being sent with weak or no permission requirements. Intents sent through the sink may be vulnerable to action-based attacks (e.g., broadcast denial of service or Activity/Service launching). If any of these Intents contain extra data, then they may also be vulnerable to eavesdropping. We issue warnings with "without data" and "with data" tags to distinguish action-based attacks from eavesdropping. Intents containing data in excess of an action, categories, component name, or package name are considered as having extra data and therefore open to both the actionand data-based attacks.

<sup>&</sup>lt;sup>1</sup>We do not consider **stopService()** as a vulnerable sink. If a Service is maliciously stopped, it can be restarted by a legitimate component when the Service is needed.

## **Component Analysis**

ComDroid's component analysis decides whether components might be susceptible to an Intent spoofing attack. ComDroid examines the application's manifest file and translates Dalvik instructions to get information about each component. For each component, Com-Droid determines whether the component is public based on the presence of Intent filters or the EXPORTED flag.

Activities and Services are always declared in the manifest. Some Receivers are also declared in the manifest. An Activity can be multiply declared in the manifest using an Activity alias, which presents an existing Activity as a separate component with its own permission, Intent filters, etc. We treat Activities and their aliases as separate components for the purpose of our analysis because an alias's fields can increase the exposure surface of the component. Also, typically one Activity is marked as a main, launching Activity that the system opens when an application is started. This Activity is public but is generally less likely to be attackable, therefore we do not issue an exposure warning for this case.

Receivers can also be dynamically created and registered using registerReceiver(BroadcastReceiver receiver, IntentFilter filter). The Intent filter is specified at registration time and can be changed each time registerReceiver is called, so we consider each registration of a Receiver as a unique component.

If a public component is protected with no permission or a weak permission, ComDroid generates a warning about a potential Intent spoofing attack (malicious Broadcast injection, malicious Activity launch, or malicious Service launch, depending on the component type). Again, we further separate the warnings into "without data" and "with data" warnings. Attacks without additional data only invoke the victim component; attacks with data additionally supply the victim component with malicious data. Both are attack surfaces, but attacks with data can potentially give an attacker more control and more opportunities to influence application state or pollute application databases. If the component receives an Intent and only reads the action, categories, component name, or package name, then it is considered to not use extra data. Otherwise, it is considered to use extra data.

ComDroid separately issues warnings for Receivers that are registered to receive system broadcast actions (actions only sent by the system). For these warnings, the solution is to add a call to android.content.Intent.getAction() to verify that the protected action is in the Intent (authenticating the sender of the Intent). This is in contrast to other Intent spoofing attacks where the more common solution is to make the component private.

ComDroid also notes when it appears that unique Intent actions are being used to communicate in place of explicit Intents. If ComDroid finds a public component that registers to receive Intents with a non-Android action string and also finds components that transmit implicit Intents with the same action string, ComDroid issues a warning. We call this "action misuse" to alert the developer that he or she may be using actions insecurely.

## Limitations and Discussion

We currently track Intent control flow across functions, but we do not distinguish between paths through if and switch statements. Instead, we follow all branches. This can lead to false negatives; e.g., an application might make an Intent explicit in one branch and implicit in another, and our tool would always identify it as explicit. In retrospect, it would have been better to track both. Additionally, our tool does not yet detect privilege delegation through pending Intents and Intents that carry URI read/write permissions; we leave this for future work. Despite these limitations, our experimental results (Section 2.5) indicate that our analysis identifies many actual application vulnerabilities.

It is important to note that ComDroid issues warnings but does not verify the existence of attacks. Some Intents and components are intentionally made public without restriction, for the purpose of inter-application collaboration. It is not possible to automatically infer the developer's intention when making a component public. We defer to the developer to examine his or her own program and verify the veracity of the warnings. ComDroid supplies the location of the potential vulnerability (filename, method, and line number), the type (malicious Activity launch, broadcast theft, etc.), and whether data leakage/injection could be involved. It could further be extended to explicitly recommend a fix for developers (e.g., make the Intent explicit).

Although ComDroid is intended primarily as a tool for developers, it takes DEX files as input instead of source code. We made this choice due to the difficulty of obtaining source code for most applications. Using DEX files, we can examine the programming practices of the most popular applications in the Android Market. The use of DEX files also allows third parties (such as anti-virus vendors) to conduct security audits. That said, ComDroid requires the user to manually investigate the warnings, which may be difficult for third parties to do quickly (especially on a large scale). Ideally, ComDroid would be used by the developers themselves or security teams contracted by the developers, since they are familiar with the code or have access to the source code.

We considered a dynamic analysis approach to ComDroid as an alternative to our static approach. A dynamic analysis tool would have the benefit of confirming a vulnerability by exploiting it at run-time (although it still may not be able to make the human distinction of whether the bug is severe or not), but it may be challenging to explore the application state space to obtain full coverage. Static analysis has the benefit of discovering vulnerabilities that may not have been exposed at runtime. It is worth investigating a combined static and dynamic tool in future research to leverage the benefits of both approaches.

| Type of Exposure                  | Percentage |
|-----------------------------------|------------|
| Broadcast Theft                   | 44 %       |
| Activity Hijacking                | 97~%       |
| Service Hijacking                 | 19 %       |
| Broadcast Injection               | 56 %       |
| System Broadcast w/o Action Check | 13~%       |
| Activity Launch                   | 57 %       |
| Service Launch                    | 14 %       |

Table 2.3: The percentage of applications that had at least one warning per exposure type

## 2.5 Inter-application Communication Evaluation

We ran ComDroid on the top 50 popular paid applications and on 50 of the top 100 popular free applications on the Android Market [3].<sup>2</sup> We report ComDroid's warning rates and discuss common application weaknesses. We emphasize that ComDroid issues warnings about potential security issues; manual review is needed to determine the functionality of the Intent or component and decide whether the exposure can lead to a severe vulnerability. We manually examined 20 applications to check ComDroid's warnings, evaluate our tool, and detect vulnerabilities.

## Automated Analysis

ComDroid detected a total of 1414 exposed surfaces across 100 applications. There were 401 warnings for exposed components and 1013 warnings for exposed Intents. In Figure 2.5 we show what fraction of sent Intents are implicit; on average, about 40% are implicit. In Figure 2.6, we show the frequency of exposed components out of the total number of components for each application, separated by component type. 50% of Broadcast Receivers are exposed, and most applications expose less than 40% of Activities to external applications. Our tool does not generate a warning for an application's primary launcher Activity; consequently, many applications that show zero exposed Activities warnings may have one public launcher Activity. There is no clear distribution for the exposure of Services because few applications have multiple Service components.

We also show the breakdown of warnings by type in Figure 2.7. Intuitively, there is more Intent communication between Activities so there are more exposure warnings for Activityrelated Intents than Broadcast- and Service-related Intents combined.

Table 2.3 shows the percentage of applications that have at least one of a given type of surface exposure. Of sending-related vulnerabilities, 44% of applications have Broadcast-

 $<sup>^{2}</sup>$ Specifically, we considered the applications ranked 51-100 for the free applications. Dedexer was not able to disassemble a few applications. In those cases, we took the next application in the list.



Figure 2.5: Histogram showing the percentage of implicit Intents out of total Intents for each application.



Figure 2.6: Histogram showing the percentage of components with warnings out of total components for each application.



Figure 2.7: Breakdown of warnings by type.

related warnings. Of these applications, none of them restrict the broadcast in any way or make the Intent explicit.

Although 97% of applications have Activity hijacking warnings, on average only 27.7% of Intents that involve starting an Activity are open to an Activity hijack. This is promising as it shows that developers are making a majority of their Activity communications explicit. 19% of applications contain Service hijacking warnings.

Over 56% of applications have a Broadcast Receiver that may be vulnerable to a Broadcast injection attack. Broken down by number of exposed Receivers per application (Figure 2.8), we see that most applications expose one or two Receivers, if any.

13% of applications have a public Receiver that accepts a system Broadcast action but does not check that the Intent actually contains that action (a definite bug that may also lead to a serious vulnerability).

57% of applications have at least one Activity that may be vulnerable to a malicious Activity launch. The other 43% only expose the main launching Activity. We display the break down of these malicious Activity launch warnings by number of malicious launch warnings per application (Figure 2.9). On average, applications have one exposed Activity in addition to the launch Activity. This is good news, as it seems that most applications are limiting their Activities from exposure. We can also see a handful of applications expose 11 to 20 Activities and can benefit from further investigation. Finally, 14% of applications have at least one Service that may be vulnerable to malicious Service launches.

Our results indicate that Broadcast- and Activity- related Intents (both sending to and receiving from) play a large role in application exposure.



Figure 2.8: Histogram showing the number of Broadcast Receivers with warnings per application.



Figure 2.9: Histogram showing the number of Activities with warnings per application.

## Manual Analysis

We randomly selected 20 applications from the 100 mentioned earlier, and then we manually inspected ComDroid's warning for these applications to evaluate how many warnings correspond to vulnerabilities. In this section, we present the findings of our manual analysis and discuss three example applications with vulnerabilities. See Appendix A.1 for the list of applications reviewed.

ComDroid generated 181 warnings for the 20 applications. We manually reviewed all of them and classified each warning as a vulnerability, not a vulnerability, or undetermined. We define a vulnerability as a component or Intent that exposes data or functionality that can be detrimental to the user. For example, an unprotected broadcast is only a vulnerability if it includes sensitive user data or if its theft results in a DoS to a legitimate service. Similarly, Activity launching is only a vulnerability if the victim Activity toggles state or operates on the Intent data in a way that negatively affects the application. Negative consequences may be context-dependent; consider an application that sends a user to another application to view a website. Normally, it will not matter if the website Activity is hijacked, but the hijacking could lead to phishing if the user expects to enter payment information at the website. We further divide vulnerabilities into two types: (1) dangerous vulnerabilities that do not rely on user interaction and (2) spoofing vulnerabilities that might occur if the user is tricked. We also separately note commonly found unintentional bugs. We classify warnings as bugs when the developer appears to be misusing or misunderstanding the Android communication design. This category includes action misuse and system broadcasts without action verification.

In order to detect vulnerabilities, we reviewed the disassembled code of the application components with warnings. We also installed the applications and interacted with them to dynamically observe their Intents. When necessary, we built "attack" code to confirm or disprove vulnerabilities. This review process does not reflect a developer's experience with ComDroid because developers would have access to the source code and knowledge of the application's intended functionality, which we did not have.

Of the 181 warnings, we discovered 20 definite vulnerabilities, 14 spoofing vulnerabilities, and 16 common, unintentional bugs (that are not also vulnerabilities). Of the 20 applications examined, 9 applications contain at least 1 definite vulnerability and 12 applications have either definite or spoofing vulnerabilities. This demonstrates the prevalence of insecure Intent communication. Table 2.4 shows the number of vulnerabilities and warnings for each category.

ComDroid has an overall vulnerability and bug detection rate of 27.6%. Broken down by Unauthorized Intent Receipt vulnerabilities/bugs and Intent Spoofing vulnerabilities/bugs, it has a rate of 22.6% and 38.6%, respectively. As shown in the table, Activity hijacking has the highest number of false positives, with a lower detection rate of 15.2%. Examining only the broadcast-related vulnerabilities (theft, injection, and system broadcasts without action check), ComDroid has a detection rate of 61.2%.

|                                 | Definite | Spoofing | Unintentional | Total  | Vuln. & Bug |
|---------------------------------|----------|----------|---------------|--------|-------------|
| Type of Exposure                | Vulns.   | Vulns.   | Bugs          | Warns. | Percentage  |
| Broadcast Theft (w/o data)      | 1        | 0        | 6             | 10     | 70.0%       |
| Broadcast Theft (w/ data)       | 2        | 0        | 2             | 4      | 100.0%      |
| Activity Hijacking (w/o data)   | 2        | 11       | 0             | 91     | 14.3%       |
| Activity Hijacking (w/ data)    | 0        | 3        | 0             | 14     | 21.2%       |
| Service Hijacking (w/o data)    | 0        | 0        | 1             | 5      | 20.0%       |
| Service Hijacking (w/ data)     | 0        | 0        | 0             | 0      |             |
| Broadcast Injection (w/o data)  | 10       | 0        | 2             | 24     | 50.0%       |
| Broadcast Injection (w/ data)   | 3        | 0        | 0             | 7      | 42.9%       |
| Sys. Broadcast w/o Action Check | 1        | 0        | 3             | 4      | 100.0%      |
| Activity Launch (w/o data)      | 0        | 0        | 0             | 9      | 0.0%        |
| Activity Launch (w/ data)       | 0        | 0        | 2             | 10     | 20.0%       |
| Service Launch (w/o data)       | 0        | 0        | 0             | 2      | 0.0%        |
| Service Launch (w/ data)        | 1        | 0        | 0             | 1      | 100.0%      |

Table 2.4: The number of vulnerabilities and bugs we found from the warnings in ComDroid.

In 25 cases, we were unable to determine whether warnings were vulnerabilities. We cannot always determine whether a surface is intentionally exposed without knowing the developer's intentions. We were uncertain of 25 of the 181 warnings. The remaining 106 warnings were false positives, i.e., not dangerous or spoofing vulnerabilities or common bugs. Of these, 6 of the warnings should not have been generated and can be attributed to shortcomings in our implementation of ComDroid. (Two Broadcast Receivers were declared without receiving methods, meaning they could not actually receive Intents. In four cases, Intents were misidentified as implicit when they were actually explicit.) The remaining 100 false positives are still exploitable attacks. However, the impact of these attacks is minor: They would be merely a nuisance to the user. For example, an Activity that turns on a "flashlight" when launched or takes some other trivial action would fall into this category. Because they represent only a nuisance, we conservatively decided not to classify them as vulnerabilities. We now discuss a few applications and the vulnerabilities we discovered in them.

**ICE:** In Case of Emergency. "ICE: In Case of Emergency" is an application that can be launched from a locked screen by a paramedic in case of an emergency [22]. It stores medical information such as medications, allergies, medical conditions, insurance information, and emergency contact information. ICE contains multiple exploitable Broadcast Receivers. One Broadcast Receiver can be used to exit any running application and lock the screen. Several of ICE's Broadcast Receivers will temporarily remove the ICE widget from the locked screen, rendering ICE unusable.

ICE's vulnerable Broadcast Receivers are accidentally public due to developer confusion over Android's complexities. Two of ICE's Receivers are registered to receive protected system broadcasts, which causes Android to make them public. ICE does not check that the received Intent has the appropriate system action, so an explicit Intent to the Receiver will trigger the same behavior as if the OS had sent the Intent. For example, ICE disappears when the operating system broadcasts an Intent with the BOOT\_COMPLETED action; a malicious application could send an explicit Intent with this action to ICE and fool it into exiting. Additionally, some of ICE's Receivers use broadcasts to pass internal notifications. The internal broadcasts have application-specific actions, e.g., com.appventive.ice.unlock\_finished. This is a misuse of action strings. These components should be made private and invoked explicitly.

**IMDb Mobile.** "IMDb Mobile" is a movie resource application [59]. It presents facts about movies, and users can look up local showtimes. IMDb's showtime Activity has buttons for the user to select a location and refresh the showtime search results. When the user clicks on one of the buttons, the Activity relays the request to a background Service. The Service responds with a public broadcast, and a Broadcast Receiver listens for the broadcast and then updates the state of the Activity's user interface. For example, the Service can send a broadcast with a com.imdb.mobile.showtimesNoLocationError action, which will cause the Activity to display an error stating that no showtimes are available for the desired location.

IMDb Mobile's broadcast Intents are intended for internal use. No other application needs to know about them, and other applications should not be able to control the user interface. However, with the current implementation, the showtime Activity can be manipulated by a malicious application. The developer should have used explicit Intents to communicate internally, and the Receiver should not be exported.

**Nationwide Bus.** Nationwide Bus is an Android application that gives bus location and arrival information for Korean cities [66]. It uses public broadcasts for internal communication. The broadcasts are used to update map and bus state. One component fetches bus information from the server and then broadcasts the data, which is intended for an internal Receiver. This is a privacy violation if the user does not want other applications to know his or her location.

Two exported components expect bus data as input. A Receiver listens for the aforementioned broadcasts, and a Service in charge of bus arrivals is started with bus data. A malicious application could send these components Intents with fake bus information, which will then be displayed in the map as fake bus stations and arrival times. The developer should have used explicit Intents for internal communication, and the Receiver and Service should not be exported.
#### **Financial Applications**

We also used ComDroid to guide a review of 10 popular financial and commercial applications. None of these applications were part of our larger set of 100 applications. The financial and commercial applications are generally well-secured; we did not find any vulnerabilities that could put the user's financial information at risk. It is clear that the applications were built to be secure, with few exposed surfaces. For example, applications make use of exported=false, use explicit Intents, and do not declare Intent filters for most of their components. It appears that the financial and commercial applications were written or reviewed by security-conscious developers, who might benefit from a tool like ComDroid during the development process.

Despite their focus on security, we found vulnerabilities in 4 of the 10 applications. One application sends an Intent to ask the browser to open up a bank website to a login page. The Intent is implicit, which puts the user at risk of an Activity hijacking phishing attack. Three applications misuse actions: they use a class name as an action string to bind to a Service. An attacker could hijack the Service and mount a denial of service attack on the parts of the application that rely on the Service. One of the vulnerable applications also contains a number of bugs that are not exploitable; it registers for a system broadcast and does not check the sender, and it uses broadcasts for internal communication. We believe that these errors, made by security-conscious developers, are indicative of the fact that Android's Intent system is confusing and tricky to use securely.

## Discussion

Our analysis shows that Android applications are often vulnerable to attack by other applications. ComDroid's warnings can indicate a misunderstanding of the Intent passing system, as we illustrated with the ICE, IMDb, and Nationwide Bus applications and can alert the developer (or a reviewer) to faulty software engineering practices that leak data and expose application internals.

Our analysis reveals that developers commonly use the action field of an Intent like an address instead of explicitly addressing the Intent. (For example, a developer might use actions prefixed with the application package name.) They add Intent filters to the components that listen for Intents with their action name, which has the undesirable side effect of making the component public. It is reasonable to assume that they are either forgetting or are not aware that they should be making their Receiver private. ComDroid can help developers be aware of surfaces that are accidentally exposed in this manner.

## Recommendations

Along with more vigilant developer coding, we also recommend changes that can be made to the Android platform to prevent unintentional exposure. One of the fundamental problems is that Intents are used for both intra- and inter- application communication and using them within an application can expose the application to external attack if the developer is not careful. Ideally, intra- and inter- application communication should be carried out through different means. Similarly, component accessibility should be divided into three categories: internal, exported to the system only, and exported to other applications.

We acknowledge that this would only prevent bugs in future applications. To fix current unintentional Intent-sending vulnerabilities in legacy applications, we propose another approach. We suggest that the system try to deliver any implicit Intents first to internal components. If the Intent can be delivered to an internal component (of the same application as the sender), it should not be delivered to any other applications. This would handle the case where developers use implicit Intents to communicate with other internal components.

In the next two sections, we detail our proposed modifications and evaluate it based on improved security and backward compatibility.

# 2.6 System Platform Changes

While the Android platform provides tools for developers to defend intra-application Intent messaging from such security vulnerabilities, we find that many developers use implicit Intents in an insecure way for application-internal messages. Specifically, developers frequently use the action field of an Intent like an address instead of explicitly addressing the Intent, exposing the communication to both unauthorized Intent receipt and Intent spoofing. Our findings motivate our recommendation for a backward-compatible change to the heuristics the platform uses to determine whether implicit Intents and components that receive them should really be exposed to other applications. Modifications to the platform immediately fix application vulnerabilities. If the onus were placed on individual developers (through a modified communication API, documentation of common communication pitfalls, better developer education, etc.), developers may not choose to update their programming practice and they may not update their legacy applications. The fix would depend on the vigilance of individual developers. As security has often not been a top priority for rapid developers, many applications may remain vulnerable. By implementing a platform-centric solution, we shift the implementation burden from individual application developers to the platform developer. Platform changes could take effect with the push of one over-the-air update and be applied to currently installed applications. Our platform changes also eliminate the need for access to application source code and avoid complex program analysis.

We propose heuristics that we implemented in Version 2.2 of the Android platform, revising and extending the changes to increase compatibility with legacy applications as we gained a better understanding of the platform. In addition, we identify ways to modify the heuristics to increase the security gain without incurring additional compatibility cost. In this section, we introduce terminology, discuss our heuristics, and discuss how Android was modified to implement our heuristics.

#### Standard actions

android.intent.action.DIAL android.intent.action.EDIT android.intent.action.MAIN android.intent.action.SEARCH android.intent.action.VIEW

Table 2.5: A few standard actions (non-exhaustive).

#### System-only broadcast actions

android.backup.intent.CLEAR android.intent.action.ACTION\_POWER\_DISCONNECTED android.intent.action.BATTERY\_CHANGED android.intent.action.REBOOT android.intent.action.TIME\_TICK

Table 2.6: A few system-only broadcast actions (non-exhaustive).

# Terminology

We define some terminology that we will be using later in this section:

- Standard action A standard action is an action string that is either (1) defined in the Android documentation or (2) used in Android applications bundled with the open-source distribution of Android, since these applications and their public APIs could be considered as convention. We identified 299 standard actions. (See Table 2.5 for examples.) Non-standard actions are any other actions (i.e., actions created by third-party developers).
- System-only broadcast Intents A system-only broadcast Intent is a special broadcast Intent that can be sent only by the operating system. Non-system processes cannot spoof these system-only broadcasts. There are 62 system-only broadcasts. (See Table 2.6 for examples of system-only broadcasts.) This is also referred to as a protected broadcast. While the operating system can send these messages, it is not limited to sending only this type of message.
- Entry-point Activity An entry-point Activity is an Activity that is the starting point of an application. Specifically, it is any Activity with an Intent Filter that receives either the MAIN action or the APPWIDGET\_CONFIGURE action, as the presence of either of these actions signifies that the Activity is intended to be started by another application.

## Heuristics

We present heuristics that the platform can use to distinguish the use of Intents for communication between components of the same application from the use of Intents between multiple applications.

**Preventing Unauthorized Intent Receipt.** If an implicit Intent can be delivered to any component in the same application, then we assume the developer intended the Intent be used for intra-application communication, and thus the Intent should not be delivered to any other application. We apply this heuristic only if the Intent contains non-standard actions. This effectively restricts modifications of expected Intent delivery to Intent actions that were uniquely created by the developer. When Intents containing these developer-created actions can be delivered within the same application, it is unlikely the Intents were intended to be exposed to other applications, as developers commonly misuse implicit Intents in this way for intra-application messaging. Also, we do not restrict delivery of broadcast Intents protected by a signature-level permission, since this type of Intent cannot be intercepted easily.

**Preventing Intent Spoofing.** Android exports a component either (1) if it has the **exported** flag, or (2) if it lacks the flag but has an Intent Filter. (Android infers that registering an Intent Filter indicates that the component is expecting external messages.)

To replace this behavior, we propose a set of more restrictive heuristics for inferring whether a component was intended to be an interface for other applications (Figure 2.10). If a component is protected with a signature-level permission, we follow the original Android behavior, so such a component that does not set the **exported** flag is exported if it contains any Intent Filters. This behavior is justified because requiring a signature-level permission already protects a component from Intent spoofing, and furthermore may indicate that the component is intended to be exposed to either system applications or applications authored by the same developer.

If, however, a component is not protected with a signature-level permission, we propose it only be exported if at least one of the following is true: it (1) sets the exported flag, (2) has an Intent Filter with a data field specified, (3) has an Intent Filter that registers to receive system-only actions, (4) is an entry-point Activity, or (5) has an Intent Filter that registers to receive Intents with a standard action. In addition, we impose the restriction that for non-entry-point Activities to be exported without the exported flag set, they must have an Intent Filter that receives the DEFAULT category.

Condition 1 identifies when a developer explicitly makes a component public or private. This flag indicates that the developer is aware of the status of the component, so we will not change the component's status contrary to the developer's explicit specification. Condition 2 specifies to make a component public when the Intent Filter contains a data field, an indicator that a sender may be trying to share data references with the external component. Condition 3 identifies a case when the component expects to receive a message from the operating system. These components must be public, sometimes subject to an additional protection discussed later. Condition 4 identifies Activities that are intended to be invoked when a user launches an application. These Activities must be exported so they can be started by external launcher applications. Condition 5 identifies components that are expecting standard actions, which may come from third-party applications, as standard actions represent a kind of Intent messaging protocol. Finally, we only export non-entrypoint Activities implicitly if they support the DEFAULT category because the standard API calls to start an Activity with an implicit Intent require the Activity to support the DEFAULT category. Without this category, the component will receive only explicit Intents. Absent this restriction, Activities that cannot typically receive Intents from external applications become vulnerable to Intent spoofing.

We also define a new **protected** property for Broadcast Receivers. If a Broadcast Receiver declares Intent Filters that *only* receive system-only broadcast actions, we export the component but flag it as *protected*, which means we enforce at runtime that only system-only broadcasts are delivered to the component. As system-only broadcasts alone match the component's Intent Filter, the only way to inject a spoofed Intent into such a component is with an explicit Intent. Enforcing the **protected** property prevents any malicious explicit Intents from reaching the component.

Finally, every time an Intent Filter is associated with a Dynamic Receiver, we created a separate **exported** flag for *each* such Intent Filter. Thus, only exported Intent Filters are considered when resolving a broadcast Intent from another application. An alternative would be to create a single **exported** flag for the Dynamic Receiver. We chose this more restrictive heuristic because developers may not be aware that associating a new Intent Filter with a Dynamic Receiver does not remove previous associations.

Note that our heuristics are more restrictive in exposing both Intents and Components than the existing Android heuristics, and as such they cannot increase the attack surface. Also, we developed these heuristics before evaluating their compatibility and security effects on applications. The only changes we made were in expanding the list of standard actions to include undocumented actions used by applications included with the open-source distribution of Android. On this basis we argue that our evaluation results generalize to other applications.

## Implementation

The relevant portions of the Android framework architecture for our heuristic changes are the system server and high-level APIs. The system server is a privileged process containing many threads that has central control over loading applications, managing their meta-data, and delivering Intents, among other things. Each Android application runs in a separate process, which has high-level APIs loaded into the address space of its instance of the Dalvik VM. An application sends and receives Intents through the high-level APIs, which in turn communicate with services running in the system server through a lower layer of IPC that marshals objects across process boundaries.



Figure 2.10: Our heuristic for when components should be exported.

The two services we modified in the system server are the Activity Manager and the Package Manager. The Activity Manager is responsible for running components, including accepting and delivering Intents. The Package Manager both loads applications and maintains their meta-data, which includes their Intent Filters, so the Package Manager resolves Intents to components.

Our implementation logs a message each time our changes differ from Android's default heuristic (i.e., an implicit Intent is prevented from escaping an application or a component is made private that would have otherwise been public in the original Android heuristic).

#### Implicit Intent Exposure Changes

To reduce the exposure of implicit Intents, we modified the Intent delivery system to try to deliver the Intent to the origin application first before trying to resolve the Intent to other applications. More specifically, we leveraged the existing setPackage(callerPackage) call which limits delivery to a specified application (effectively making the implicit intent temporarily application-explicit) and modified the Activity Manager to call it on any implicit

| Component Type                        | Methods Modified                     |  |
|---------------------------------------|--------------------------------------|--|
| Service (Context class)               | <pre>startService(),</pre>           |  |
|                                       | <pre>bindService(),</pre>            |  |
|                                       | <pre>stopService()</pre>             |  |
| Receiver (Context class)              | <pre>sendBroadcast(),</pre>          |  |
|                                       | <pre>sendOrderedBroadcast(),</pre>   |  |
|                                       | <pre>sendStickyBroadcast(),</pre>    |  |
|                                       | sendStickyOrderedBroadcast(),        |  |
|                                       | <pre>removeStickyBroadcast()</pre>   |  |
| Activity (Context and Activity class) | <pre>startActivity(),</pre>          |  |
|                                       | <pre>startActivityForResult(),</pre> |  |
|                                       | <pre>startActivityIfNeeded(),</pre>  |  |
|                                       | <pre>startActivityFromChild()</pre>  |  |

Table 2.7: A list of the Intent sending methods that were modified.

Intent (with the destination set to the origin application) before attempting to resolve it to a component through the Package Manager. If the resolution fails, then there must be no application-internal component that can respond to the Intent, so we call setPackage(null) to make the Intent implicit again and attempt to resolve it once more.

To utilize setPackage(callerPackage), we had to modify the implementation of the Intent sending APIs to pass the name of the calling Android package name, a string that uniquely identifies each application, to the Activity Manager. This is necessary because otherwise the Activity Manager can learn only the calling application's UID, PID, and the primary application associated with its process. Since multiple applications can share a process and UID, this information is insufficient to identify the calling application. We modified the implementation of the sending methods, not the interfaces, so this change does not affect the API for developers. We list those methods in Table 2.7.

The PackageManager class also provides methods for resolving Intents to components without sending anything. These are queryBroadcastReceivers(), queryIntentActivities(), resolveService(), queryIntentServices(), resolveActivity(), and query-IntentActivityOptions(). We modified all of these in the same manner.

For broadcasts, one challenge we encountered was the lack of an interface for resolving an Intent to a list of Dynamic Receivers that were created in a specified application. (Due to a bug in the Android source code, **setPackage(**) does not limit the recipient to specific applications for broadcast Intents.) We created separate lists for internal Broadcast Receivers and Dynamic Receivers, and if the Intent does not match either kind of receiver, we attempt resolution to external receivers.

#### **Component Exposure Changes**

To implement our component exposure heuristic, we added functionality to help protect Broadcast Receivers and Dynamic Receivers that only expect system-only Broadcasts (from explicit Intent spoofing attacks).

We implemented the enforcement of the protected property in the Activity Manager. If a broadcast Intent resolves to a protected Broadcast Receiver, we allow the Intent to be delivered if the caller has the capability to send a system-only broadcast (i.e., it is one of the operating system processes). Otherwise, we ask the Package Manager whether the Intent's action is system-only. If it is, we log the error and prevent delivery to the Broadcast Receiver.

Implementing the heuristic for Dynamic Receivers was more complex. As a Dynamic Receiver can register multiple Intent Filters, each Intent Filter needs state to track its exposure status. First, we added an exported field to the BroadcastFilter class, which represents a single Intent Filter. Second, we implemented the code to set the exported field using our heuristic for each call to registerReceiver() in the Activity Manager. Finally, we added code to enforce the exported property in the Activity Manager. If a BroadcastFilter is not exported, we check whether the caller UID and the UID of the application that registered the filter match. If they do not, we log the error and skip the current Dynamic Receiver.

# 2.7 Platform Evaluation

We evaluated our proposed changes on a collection of 969 popular (top free and paid) applications from the Android Market.<sup>3</sup> We believe this to be a suitable dataset as popular applications are more likely to be on users' phones, representing a realistic approximation of potential application interaction. With this dataset, we built upon ComDroid to look for specific instances where our changes prevent intentional inter-application communication, contributing to incompatibility, as well as instances where our changes eliminate ComDroid vulnerability warnings, contributing to increased security. We call this new tool IntraComDroid. (Hereafter we use IntraComDroid to refer to the tool that we use for our compatibility analysis and component and Intent modification tracking and we use ComDroid to refer to the tool that produces all message-related vulnerability warnings.)

In addition, we used IntraComDroid to examine the extent to which our changes fix concrete security vulnerabilities and unintentional, unnecessarily exposed Intents and components we previously identified. We reexamine a case study of a bus schedule application with multiple security vulnerabilities and find that our changes patch all the vulnerabilities.

<sup>&</sup>lt;sup>3</sup>We originally started with 1,000 applications and removed applications from the dataset that only consisted of keys to unlock paid features for free applications or were duplicates.

# **Compatibility Analysis**

We used static analysis to guide our compatibility investigation. To identify situations where our changes may break inter-application communication, IntraComDroid resolves and records all messages each application receives and sends. Then it analyzes all messages and receiving components in the set to determine the pairs of applications that can communicate with one another. It also analyzes each application and flags all cases in which our proposed heuristics would change the exposure of an Intent or component. Using the previous analysis of all communication in the dataset, IntraComDroid logs two types of potential incompatibilities that warrant manual examination.

The first are instances where an application sends an Intent that one of its own components can receive but components in other applications can also receive. If such a case is an instance of intentional inter-application communication, our changes may break compatibility, as they prevent the Intent from being delivered to the other application.

The second are instances where our changes make a component private, but where IntraComDroid either found other applications that could send Intents that could be received by the component or where it could not find any Intents that address the component at all. In these cases, the concern is that these components were intended as public APIs that our changes will break or that there was an error in the analysis and an Intent was not properly identified.

We manually analyzed each list of potential incompatibilities using several methods. We:

- searched for documentation of public Intent APIs to confirm intentional inter-application communication
- checked archives of Android applications to see whether two applications were different versions of the same application, and presumably not communicating
- read disassembled code to find undetected, internal Intent senders and to understand how Intents were being used in applications
- ran the applications on our modified Android platform, attempting to trigger breakage

Intent exposure compatibility. Out of 969 applications each checked against all 968 other applications, we found 99.4% are compatible with our proposed changes to implicit Intent exposure (Table 2.8). We classify the six incompatible applications into two categories and show that both can be fixed easily in either application code, in the platform, or in both. First, four applications broadcast Intents to other applications, but also declare they themselves can handle the Intent. In the case where all of these applications are by the same developer, simply protecting the broadcasts with a signature-level permission declared in all applications resolves the incompatibility. If there is no restriction on who developed the receiving applications, we call this a *broadcast protocol* and propose fixing the incompatibility by adding a flag to broadcasts that makes them explicitly public (implemented by application

|               | Intent           | Component        |
|---------------|------------------|------------------|
|               | Exposure Changes | Exposure Changes |
| Apps Analyzed | 969              | 100              |
| Compatible    | 99.4%            | 93.0%            |
| Incompatible  | 0.6%             | 5.0%             |
| Uncertain     | 0.0%             | 2.0%             |

Table 2.8: Compatibility analysis results

developers). In the second category, four applications share common Service code between two applications by the same developer. Since both applications have the same developer, the developer can resolve the incompatibility by simply protecting the Services with a Signaturelevel permission declared in both applications.

**Component exposure compatibility.** Out of 100 of the most popular applications each checked against all 968 other applications, we found 93% were compatible with our proposed changes to the heuristics used to export components. We were unable to determine whether our changes are incompatible with two applications. We found five incompatible applications, which fell into two types, both of which can be easily fixed in application code. First, two applications use third-party libraries based on Intents for inter-application communication. In this case, only the library developer need document how to explicitly export the appropriate components. New documentation would provide compatibility for new applications, while simply exporting the right components to be exported for inter-application communication between applications developed by the same party. The incompatible components were all Receivers, so protecting the broadcasts with a Signature-level permission declared in all applications would make the applications compatible.

# Security Analysis

We evaluated our proposed heuristics by examining the extent to which our changes concretely increase application security for the 20 applications we manually analyzed previously. We find our applied heuristics would patch 100% of the subset of warnings that were marked as intra-application communication. Of all of the vulnerabilities and bugs that were detected with ComDroid, our new heuristics patch 31.4% (11/35) of the security vulnerabilities and 100.0% (15/15) of the unintentional exposures. We examined the unpatched vulnerabilities and bugs, and they are all in the class of vulnerabilities where external communication is intended (but still vulnerable to third-party attack). Of the 17 remaining unauthorized Intent receipt vulnerabilities and bugs, 4 could be fixed by adding a requirement that certain Intents can be received only by system applications (e.g., Intents that send android.settings.INPUT\_METHOD\_SETTINGS or android.intent.action.DELETE). This is

|             | Component Exposure | Intent Exposure |
|-------------|--------------------|-----------------|
| Total Warn- | 3182               | 8680            |
| ings        |                    |                 |
| Eliminated  | 1431 (45.0%)       | 1608 (18.5%)    |
| Warnings    |                    |                 |

Table 2.9: The proportion of ComDroid warnings eliminated by our heuristics.

the reverse of the existing restriction that some Intents can be sent only by the system. However, this is outside of the scope of our work. Of the 7 unpatched Intent spoofing vulnerabilities, 6 could be fixed by making some of the actions the components expect system-only actions (e.g., android.intent.action.TIME\_SET, android.appwidget.action.APPWIDGET\_ UPDATE, android.provider.Telephony.SMS\_RECEIVED). With these changes, our heuristic would then identify them as protected Receivers which would be protected by our system.

We also evaluated our proposed heuristics by examining the proportion of potential security vulnerabilities detected by ComDroid that our changes would eliminate in the set of 969 applications. We use the term "potential security vulnerabilities" because ComDroid issues warnings for exposed communication. Manual examination is required to identify a vulnerability, which we classify as something that exposes data or functionality that can be detrimental to the user, so vulnerabilities are context-dependent. For example, gaining control of an Activity is not considered an exploitable vulnerability unless it could lead to theft of payment or password information.

We find that our platform changes would eliminate 45.0% and 18.5% of ComDroid's receiving and sending warnings, respectively (Table 2.9). While we can only speculate on how many concrete vulnerabilities this is, we do know that these changes make up 25.6% of the total warnings. This means that developers using ComDroid have 25.6% fewer warnings that they would have had to examine otherwise and may reduce vulnerabilities by 25.6% as well.

Finally, we revisit a case study on Nationwide Bus, an application that provides bus location and arrival information for Korean cities [66]. Three kinds of security vulnerabilities were found in the application. First, the application uses an implicit broadcast Intent to send bus information to its own Broadcast Receiver, exposing privacy-sensitive information to eavesdropping applications. Second, the receiver is exported, exposing it to malicious injection of false bus stops and schedules. Third, the receiver forwards the bus information to an exported Service, which is also exposed to malicious injection of false information.

This application illustrates precisely the attack surfaces our new heuristic aims to reduce. As the application can receive its own broadcast, our heuristic detects the Intent as intra-application and prevents eavesdropping applications from receiving it. Furthermore, since our heuristic for component exposure makes the affected components private, other applications can no longer inject malicious information into the components.

## Discussion

We discuss the limitations of our approach, alternative implementations, and the implications for future systems.

**Limitations.** To prevent unauthorized Intent receipt, our heuristics prevent an implicit Intent from being delivered to external applications when the originating application can receive the Intent. This heuristic restricts an application from sending an Intent to both internal and external recipients. However, our evaluation shows this is currently not a common use of Intents. If this is judged to be an important use case for Intents, a future API could accept a flag explicitly allowing Intents to be delivered externally.

Our compatibility evaluation is limited by the analysis used by IntraComDroid. If IntraComDroid fails to correctly resolve the contents of an Intent for external communication, we lose knowledge of the sent Intent, and thus miss a possible breakage (if our heuristic also makes the recipient component private). For example, static analysis cannot determine the contents of Intents that are dynamically received and forwarded to other components. Similarly, we are also limited by the size of our dataset. If an application communicates with an application outside of our dataset, we have no knowledge of what that other application does. This could result in false negatives, which may cause us to underestimate the compatibility cost of our changes. Despite these limitations, we believe the dataset size and ComDroid analysis are sufficient to estimate compatibility and security.

An alternative approach to static analysis is to run the applications dynamically. The limitation here is that current dynamic Android analyzers do not achieve sufficient execution coverage. They may fail to trigger specific events required to construct and send an Intent, thereby losing outgoing message information. Also, to evaluate whether a message should be delivered internally or externally, the system would need to have knowledge of all possible receivers for all applications. Due to device resource constraints, it is impossible to install a large set of applications on the device at once. Alternatively, each pair of applications in a set could be installed iteratively, but this technique is slow. Our static approach achieves reasonable code coverage and is not limited by resource constraints.

Our compatibility evaluation is also limited by our choice of the most popular applications without regard to what applications have common developers. Since applications with a common developer are more likely to communicate with each other, considering them separately would increase the relevance of a future compatibility evaluation.

Alternative Implementations. In the case that Android is hesitant to push these changes to the platform, our proposed heuristic can be applied in other ways. We discuss the alternatives in order of decreasing security and increasing compatibility. First, the modified platform would be distributed as a third-party, custom ROM, like CyanogenMod [6]. This approach would maintain our compatibility rates, but only users who choose that platform would gain any security benefit. Second, Android Market and other markets could use our static analysis tool to identify applications where our changes could break compatibility. Then they could use our heuristics to selectively perform binary rewriting (to make Intents explicit and components private) on only the vulnerable but compatible applications. This would increase compatibility and security on most applications. Third, anti-virus software could use our heuristics to identify intra-application communication. It could monitor communication, issue alerts for external communication with "internal" Intents or components, and ignore any known broadcast protocols.

Finally, our heuristics could be used in a lint tool to detect exposed intra-application communication. The lint tool could warn the developer and provide a recommended remediation (e.g., set the **exported** flag to false) any time our heuristics would treat a component differently than the Android platform does. The false positive rate would be low (the recommended remediation would be problematic for only 0.6%-7% of applications, as our compatibility evaluation shows), and if developers test their applications, false positives would likely be relatively harmless. This approach achieves full compatibility, but security gains would accrue gradually over time (as developers would have to opt-in).

**Implications for Future Systems.** Guessing developer intention is the primary difficulty in automatically patching intra-application communication vulnerabilities. Although our approach shows a low compatibility cost, one way to avoid the guesswork is to make developers declare their intentions upfront. We recommend that future systems require developers to make their intentions explicit.

As a lesson for future systems, we recommend that a system should not implicitly open holes in isolation and expose applications to possible attack without an explicit request by the developer. Isolation should be enabled by default, and system designers should be wary of complex mechanisms that make it tricky for developers to predict when their application might be exposed to attack. Android violates this principle: under certain conditions, it will treat a component as exported even if the **exported** flag has not been set. This has misled developers into using Intents in an unsafe way. Unfortunately, many applications have already been written assuming this behavior, so it cannot be easily changed. In this work, we develop intricate measures to reduce the number of applications put at risk, while striving to maintain a high level of backward compatibility for existing applications. However, future systems could avoid this complexity and avoid compatibility problems by simply following this principle from the start. In particular, we recommend future systems provide different APIs to separate internal communication from external communication.

Android could follow this recommendation in a future API revision by making the **exported** flag mandatory and adding separate API calls for sending Intents to internal and external components.

# 2.8 Other Mobile Platforms

Windows Phone 7 (WP7) and iOS also provide third-party application platforms. However, their inter-application communication systems are less complex than Android's. WP7 appli-

cations can only send messages to a small number of trusted system applications (e.g., the browser); third-party applications cannot receive messages. Consequently, we are not aware of any security risks associated with WP7 application communication.

iOS applications can choose to accept inter-application messages by registering custom URI schemes with the OS. When an application receives a message, the application is opened and moved to the foreground. (For example, sending a message to skype://15554446666 opens the Skype application.) This is the equivalent of sending an Intent to start an Activity, and a malicious iOS application could mount an Activity hijacking attack by registering for another application's scheme. However, the remaining Android communication attacks are not applicable to iOS. iOS developers are unlikely to accidentally expose functionality because schemes are only used for public interfaces; different types of messages are used for internal communication. Our recommendations for Android (Section 2.5) aim to create the same distinction between internal and external messages in Android.

# 2.9 Related Work

Attack Surfaces. The concept of examining systems to identify and quantify their attack surfaces is not new. Metrics have been proposed for evaluating the exposed attack surface of a generic system [58], and attack surface reduction has widely been recognized as an approach to improving system security [71].

**Non-mobile Systems.** The Android inter-application communication system is analogous to a (local) network system. As such, it must deal with standard threats that apply to all messaging systems, for example, eavesdropping, spoofing, denial of service, etc. [36]. As we have shown in this chapter, these threats are present in Android's Intent system.

Problems in Android's communication model are similar to problems with decentralized information flow control (DIFC) in other systems. DIFC lets applications explicitly express their information flow policies (i.e., which applications communicate and how they communicate) to the OS or a language runtime, which then enforces the policies [42, 65, 73, 91]. The problems that arise in Android relate to developers' difficulty with setting appropriate communication policies; the same problems exist in DIFC models, which also require the developer to write policies.

Application Communication. We are not the first to realize that Android developers make mistakes that can compromise security. Burns [34] discusses common developer errors, such as using Intent filters instead of permissions. He recommends using permissions to protect components and validate caller identity. Our work builds on these concepts and provides a tool for detecting these errors. Similarly, Enck et al. [47] examine Android security policies and discuss some developer pitfalls. They present a decompiler to recover application source code from DEX files and apply COTS Java static analysis tools to the source code to examine various properties in applications. They investigate how broadcast Intents can leak information and how information can be injected into Receivers [45]. Their investigation of broadcast Intents and Receivers is limited to data-based attacks, and they do not discuss attacks involving Activities or Services. We present non-data attacks on Receivers (e.g., denial of service attacks and state change) and extensively consider Activities and Services.

SCanDroid [53] is a static analysis tool that takes a data-centric approach to reasoning about the consistency of security specifications. It analyzes data policies in application manifests and data flows across Content Providers. Based on its analysis, it makes a recommendation on whether an application can be installed with the permissions it has without violating the permissions of other applications. Used together, ComDroid and SCanDroid could combine surface exposure with database permission violations. However, SCanDroid currently requires users to have access to application source code, which may not be feasible.

TaintDroid [44] provides system-wide dynamic taint tracking for Android. It discovered 68 potential information misuse examples in 20 applications. Unlike ComDroid, TaintDroid focuses solely on data flow and does not consider action-based vulnerabilities. We found many control-flow vulnerabilities using ComDroid. Also, TaintDroid is meant to be a post-production tool for real-time analysis, while ComDroid can be used as either a pre- or post-production tool. TaintDroid and ComDroid are complementary tools.

Insecure application communication and exposure can lead to other attacks in addition to information leakage, information injection, and component hijacking. Maji et al. measure the robustness of the Intent system against malformed or unexpected Intents [?]. They build a tool, JJB, to fuzz test Android components. They find that input validation and exception handling are overlooked problems whose absence can result in crashes of the Android runtime system.

Unrestricted access to components can also lead to privilege escalation. Many researchers have examined this problem [39, 41, 52, 55]. Davi et al. discuss privilege escalation through the Android Scripting Environment [39] and Grace et al. present a static analysis tool to detect such attacks [55]. Felt et al. [52] and Dietz et al. [41] further propose runtime defenses. By making unintentionally exposed components private, our work can prevent some access by third-party Intents, thereby avoiding a portion of these problems.

# 2.10 Conclusion

While the Android message passing system promotes the creation of rich, collaborative applications, it also introduces the potential for attack if developers do not take precautions. We examine inter-application communication in Android and present several classes of potential attacks on applications. Outgoing communication can put an application at risk of Broadcast theft (including eavesdropping and denial of service), data theft, result modification, and Activity and Service hijacking. Incoming communication can put an application at risk of malicious Activity and Service launches and Broadcast injection.

We provide a tool, ComDroid, that developers can use to find these kinds of vulnerabilities. Our tool relies on DEX code, so third parties or reviewers for the Android Market can use it to evaluate applications whose source code is unavailable. We analyzed 100 applications and verified our findings manually with 20 of those applications. Of the 20 applications, we identified 12 applications with at least one vulnerability. This shows that applications can be vulnerable to attack and that developers should take precautions to protect themselves from these attacks.

We also showed that developer confusion can be attributed to making applications vulnerable to attack. We further describe an implementation of a better heuristic in the Android platform for detecting unintentional inter-application Intent messaging. We showed that our proposal reduces the number of such vulnerabilities. We evaluated both the security gain and the compatibility cost of our proposed changes, finding 99.4% and 93.0% of applications analyzed are compatible with our Intent exposure and component exposure changes, respectively. Our proposal fixes 31.4% of security flaws found in a previous study. Our work suggests that intra-application communication vulnerabilities in applications can be patched by the Android platform in a way that is reasonably backward-compatible with existing applications.

# 2.11 Acknowledgments

This work appeared in part in MobiSys 2011 [37] and SPSM 2012 [62]. We thank Adrienne Felt and Kate Greenwood for their help with manually analyzing results from ComDroid. We thank David Kantola for his work on modifying the Android platform and manually analyzing results from IntraComDroid and Warren He for his work on manually analyzing results from IntraComDroid.

# Chapter 3

# Analyzing Android Permissions

# 3.1 Introduction

Android supports third-party development with an extensive API that provides applications with access to phone hardware (e.g., the camera), WiFi and cellular networks, user data (e.g., received text messages), and phone settings. Access to privacy- and security-relevant parts of Android's rich API is controlled by an install-time application permission system. Each application must declare upfront what permissions it requires, and the user is notified during installation about what permissions it will receive. If the user does not want to grant a permission to an application, she can cancel the installation process.

Install-time permissions can provide users with privacy control and reduce the impact of bugs and vulnerabilities in applications. However, an install-time permission system is ineffective if developers routinely request more permissions than they require. Overprivileged applications expose users to unnecessary permission warnings and increase the impact of a bug or vulnerability. We study Android applications to determine whether Android developers follow least privilege or overprivilege their applications.

We present a tool, STOWAWAY, that detects overprivilege in compiled Android applications. Stowaway is composed of two parts: a static analysis tool that determines what API calls an application makes and a permission map that identifies what permissions are needed for each API call. Android's documentation does not provide sufficient permission information for such an analysis, so we empirically determined Android 2.2's access control policy. Using automated testing techniques, we achieved 85% coverage of the Android API. Our permission map provides insight into the Android permission system and enables us to identify overprivilege.

We apply Stowaway to 940 Android applications from the Android Market and find that about one-third of applications are overprivileged. The overprivileged applications generally request few extra privileges: more than half only contain one extra permission, and very few request more than four unnecessary permissions. We investigate causes of overprivilege and find that many developer errors stem from confusion about the permission system. Our results indicate that developers are trying to follow least privilege, which supports the potential effectiveness of install-time permission systems like Android's.

Android provides developer documentation, but its permission information is limited. The lack of reliable permission information may cause developer error. The documentation lists permission requirements for only 78 methods, whereas our testing reveals permission requirements for 1,207 methods (a fifteen-fold improvement over the documentation). Additionally, we identify 6 errors in the Android permission documentation. This imprecision leaves developers to supplement reference material with guesses and message boards. Developer confusion can lead to overprivileged applications, as the developer adds unnecessary permissions in an attempt to make the application work correctly.

**Contributions.** We provide the following contributions:

- 1. We developed STOWAWAY, a tool for detecting overprivilege in Android applications. We evaluate 940 applications from the Android Market with Stowaway and find that about one-third are overprivileged.
- 2. We identify and quantify patterns of developer error that lead to overprivilege.
- 3. Using automated testing techniques, we determine Android's access control policy. Our results represent a fifteen-fold improvement over the documentation.

Other existing tools [45,46] and future program analyses could make use of our permission map to study permission usage in Android applications.

**Organization.** Section 3.2 provides an overview of Android and its permission system, Section 3.3 discusses our API testing methodology, and Section 3.4 describes our analysis of the Android API. Finally, Section 3.5 describes our static analysis tools for detecting overprivilege and Section 4.4 discusses our application overprivilege analysis.

# 3.2 The Android Permission System

Android has an extensive API and permission system. We first provide a high-level overview of the Android application platform and permissions. We then present a detailed description of how Android permissions are enforced.

# Android Background

Android smartphone users can install third-party applications through the Android Market [3] or Amazon Appstore [20]. The quality and trustworthiness of these third-party applications vary widely, so Android treats all applications as potentially buggy or malicious. Each application runs in a process with a low-privilege user ID, and applications can access only their own files by default. Applications are written in Java (possibly accompanied by native code), and each application runs in its own virtual machine.

Android controls access to system resources with install-time permissions. Android 2.2 defines 134 permissions, categorized into three threat levels:

- 1. *Normal* permissions protect access to API calls that could annoy but not harm the user. For example, SET\_WALLPAPER controls the ability to change the user's background wallpaper.
- 2. *Dangerous* permissions control access to potentially harmful API calls, like those related to spending money or gathering private information. For example, Dangerous permissions are required to send text messages or read the list of contacts.
- 3. Signature/System permissions regulate access to the most dangerous privileges, such as the ability to control the backup process or delete application packages. These permissions are difficult to obtain: Signature permissions are granted only to applications that are signed with the device manufacturer's certificate, and SignatureOrSystem permissions are granted to applications that are signed or installed in a special system folder. These restrictions essentially limit Signature/System permissions to pre-installed applications, and requests for Signature/System permissions by other applications will be ignored.

Applications can define their own permissions for the purpose of self-protection, but we focus on Android-defined permissions that protect system resources. We do not consider developer-defined permissions at any stage of our analysis.

Permissions may be required when interacting with the system API, databases, and the message-passing system. The public API [21] describes 8,648 methods, some of which are protected by permissions. User data is stored in *Content Providers*, and permissions are required for operations on some system Content Providers. For example, applications must hold the READ\_CONTACTS permission in order to execute READ queries on the Contacts Content Provider. Applications may also need permissions to receive *Intents* from the operating system. System Intents are messages that notify applications of events, such as a change in network connectivity, and some system Intents are delivered only to applications with appropriate permissions. Furthermore, permissions are required to send Intents that mimic the contents of system Intents.

# **Permission Enforcement**

We describe how the system API, Content Providers, and Intents are implemented and protected. To our knowledge, we are the first to describe the Android permission enforcement mechanisms in detail.

## The API

**API Structure.** The Android API framework is composed of two parts: a library that resides in each application's virtual machine and an implementation of the API that runs in the system process. The API library runs with the same permissions as the application it accompanies, whereas the API implementation in the system process has no restrictions. The library provides syntactic sugar for interacting with the API implementation. API calls



Figure 3.1: The architecture of the Android platform. Permission checks occur in the system process.

that read or change global phone state are proxied by the library to the API implementation in the system process.

API calls are handled in three steps (Figure 3.1). First, the application invokes the public API in the library. Second, the library invokes a private proxy interface, also in the library. Third, the library's private interface uses inter-process communication to ask a service running in the system process to perform the desired operation. For example, if an application calls ClipboardManager.getText(), the call will be relayed to IClipboard, which proxies the call to the system process's ClipboardService.

An application can use Java reflection [72] to access all of the API library's hidden and private classes, methods, and fields. Some private interfaces do not have any corresponding public API; however, applications can still invoke them using reflection. These non-public library methods are intended for use by Google applications or the framework itself, and developers are advised against using them because they may change or disappear between releases [56]. Nonetheless, some applications do use them anyway. Code running in the system process is separate and therefore immune to reflection.

**Permissions.** To enforce permissions, various parts of the system invoke a permission validation mechanism to check whether a given application has a specified permission. The permission validation mechanism is implemented as part of the trusted system process, and invocations of the permission validation mechanism are spread throughout the API. There is no centralized policy for checking permissions when an API is called. Rather, mediation is contingent on the correct placement of permission validation calls.

When invoked, the API implementation in the system process calls the permission validation mechanism to check that the invoking application has the necessary permissions. In some cases, the API library may also redundantly check these permissions, but such checks cannot be relied upon: applications can circumvent them by directly communicating with the system process.

A small number of permissions are enforced by Unix groups, rather than the Android permission validation mechanism. In particular, when an application is installed with the INTERNET, WRITE\_EXTERNAL\_STORAGE, or BLUETOOTH permissions, it is assigned to a Linux group that has access to the pertinent sockets and files. Thus, the Linux kernel enforces the access control policy for these permissions. The API library (which runs with the same rights as the application) can accordingly directly operate on these sockets and files, without needing to invoke the API implementation in the system process.

**Native Code.** Applications can include native code in addition to Java code, but native code is still beholden to the permission system. Attempts to open sockets or files are mediated by Linux permissions. Native code cannot communicate directly with the system API. Instead, the application must create Java wrapper methods to invoke the API on behalf of the native code. Android permissions are enforced as usual when the API calls are executed.

#### **Content Providers**

System Content Providers are installed as standalone applications, separate from the system process and API library. The system places restrictions on its Content Providers the same way that applications place restrictions on their own Content Providers. The system Content Providers are protected with both static and dynamic permission checks.

Static declarations assign separate read and write permissions to a given Content Provider; by default, these permissions are applied to all resources stored by the Content Provider. Restrictions are also applied at a finer granularity by associating permissions with a path (e.g., content://a/b), or by specifying grantUriPermissions in the Content Provider's declaration to override permission enforcement for a specific path. For example, a Content Provider that stores both public and private notes might want to set a default permission requirement for the whole Content Provider, but then allow unrestricted access to the public notes. Extra permission requirements can similarly be set for certain paths, making data under those paths accessible only if the calling application has the default permissions for the provider as well as the path-specific permissions.

Content Providers can also enforce permissions programmatically: the Content Provider code that handles a query can explicitly call the system's permission validation mechanism to require certain permissions. This gives the developer greater control over the granularity of the permission enforcement mechanism, allowing her to selectively require permissions for query values or database data.

#### Intents

Android's Intent system is used extensively for inter- and intra-application communication. Applications may restrict who can receive an Intent by attaching a permission requirement to the Intent before sending it [47]. The OS uses the same mechanism to restrict who may receive its Intents.

To prevent applications from mimicking system Intents, Android restricts who may send certain Intents. All Intents are sent through the ActivityManagerService (a system service), which enforces this restriction. We found two distinct cases that restrict the sending of system Intents. Some system Intents can only be sent by applications with appropriate permissions. Other system Intents can only be sent by processes whose UID matches the system's. Intents in the latter category cannot be sent by applications, regardless of what permissions they hold, because these Intents must originate from the system process.

# 3.3 Permission Testing Methodology

Android's access control policy is not well-documented, but it is necessary to determine whether applications are overprivileged. To address this shortcoming, we developed techniques to empirically determine the access control policy that Android enforces. We use testing to construct a permission map that identifies the permissions required for each method in the Android API. In particular, we modified Android 2.2's permission verification mechanism to log permission checks as they occur. We then generated unit test cases for API calls, Content Providers, and Intents. Executing these tests allowed us to observe the permissions required to interact with system APIs. A core challenge was to build unit tests that obtain call coverage of all platform resources.

# The API

As described in §3.2, the Android API provides applications with a library that includes public, private, and hidden classes and methods. The set of private classes includes proxy interfaces to the system services. All of these classes and methods are accessible to applications using Java reflection, so we must test them to identify permission checks. We conducted testing in three phases: feedback-directed testing; customizable test case generation; and manual verification.

#### Feedback-Directed Testing

For the first phase of testing, we used Randoop, an automated, feedback-directed, objectoriented test generator for Java [76,78]. Randoop takes a list of classes as input and searches the space of possible sequences of methods from these classes. We modified Randoop to run as an Android application and to log every method it invokes. Our modifications to Android log every permission that is checked by the Android permission validation mechanism, which lets us deduce which API calls trigger permission checks.

Randoop searches the space of methods to find methods whose return values can be used as parameters for other methods. It maintains a pool of valid initial input sequences and parameters, initially seeded with primitive values (e.g., int and String). Randoop builds test sequences incrementally by randomly selecting a method from the test class's methods and selecting sequences from the input pool to populate the method's arguments. If the new sequence is unique, then it is executed. Sequences that complete successfully (i.e., without generating an exception) are added to the sequence pool. Randoop's goal is full coverage of the test space. Unlike comparable techniques [23, 38, 77], Randoop does not need a sample execution trace as input, making large-scale testing such as API fuzzing more manageable. Because Randoop uses Java reflection to generate the test methods from the supplied list of classes, it supports testing non-public methods. We modified Randoop to also test nested classes of the input classes.

Limitations. Randoop's feedback-guided space exploration is limited by the objects and input values it has access to. If Randoop cannot find an object of the correct type needed to invoke a method in the sequence pool, then it will never try to invoke the method. The Android API is too large to test all interdependent classes at once, so in practice many objects are not available in the sequence pool. We mitigated this problem by testing related classes together (for example, Account and AccountManager) and adding seed sequences that return common Android-specific data types. Unfortunately, this was insufficient to produce valid input parameters for many methods. Many singleton object instances can only be created through API calls with specific parameters; for example, a WifiManager instance can be obtained by calling android.content.Context.getSystemService(String) with the parameter "wifi." We addressed this by augmenting the input pool with specific primitive constants and sequences. Additionally, some API calls expect memory addresses that store specific values for parameters, which we were unable to solve at scale.

Randoop also does not handle ordering requirements that are independent of input parameters. In some cases, Android expects methods to precede each other in a very specific order. Randoop only generates sequence chains for the purpose of creating arguments for methods; it is not able to generate sequences to satisfy dependencies that are not in the form of an input variable. Further aggravating this problem, many Android methods with underlying native code generate segmentation faults if called out of order, which terminates the Randoop testing process.

#### **Customizable Test Case Generation**

Randoop's feedback-directed approach to testing failed to cover certain types of methods. When this happened, there was no way to manually edit its test sequences to control sequence order or establish method pre-conditions. To address these limitations and improve coverage, we built our own test generation tool. Our tool accepts a list of method signatures as input, and outputs at least one unit test for each method. It maintains a pool of default input parameters that can be passed to methods to be called. If multiple values are available for a parameter, then our tool creates multiple unit tests for that method. (Tests are created combinatorially when multiple parameters of the same method have multiple possible values.) It also generates tests using null values if it cannot find a suitable parameter. Because our tool separates test case generation from execution, a human tester can edit the test sequences produced by our tool. When tests fail, we manually adjust the order of method calls, introduce extra code to satisfy method pre-conditions, or add new parameters for the failing tests.

Our test generation tool requires more human effort than Randoop, but it is effective for quickly achieving coverage of methods that Randoop was unable to properly invoke. Overseeing and editing a set of generated test cases produced by our tool is still substantially less work than manually writing test cases. Our experience with large-scale API testing was that methods that are challenging to invoke by feedback-directed testing occur often enough to be problematic. When a human tester has the ability to edit failing sequences, these methods can be properly invoked.

#### Manual Verification

The first two phases of testing generate a map of the permission checks performed by each method in the API. However, these results contain three types of inconsistencies. First, the permission checks caused by asynchronous API calls are sometimes incorrectly associated with subsequent API calls. Second, a method's permission requirements can be argument-dependent, in which case we see intermittent or different permission checks for that method. Third, permission checks can be dependent on the order in which API calls are made. To identify and resolve these inconsistencies, we manually verified the correctness of the permission map generated by the first two phases.

We used our customizable test generation tool to create tests to confirm the permission(s) associated with each API method in our permission map. We carefully experimented with the ordering and arguments of the test cases to ensure that we correctly matched permission checks to asynchronous API calls and identified the conditions of permission checks. When confirming permissions for potentially asynchronous or order-dependent API calls, we also created confirmation test cases for related methods in the pertinent class that were not initially associated with permission checks. We ran every test case both with and without their required permissions in order to identify API calls with multiple or substitutable permission requirements. If a test case throws a security exception without a permission but succeeds with a permission, then we know that the permission map for the method under test is correct.

Testing The Internet Permission. Applications can access the Internet through the Android API, but other packages such as java.net and org.apache also provide Internet access. In order to determine which methods require access to the Internet, we scoured the documentation and searched the Internet for any and all methods that suggest Internet access. Using this list, we wrote test cases to determine which of those methods require the INTERNET permission.

# **Content Providers**

Our Content Provider test application executes query, insert, update, and delete operations on Content Provider URIs associated with the Android system and pre-installed applications. We collected a list of URIs from the android.provider package to determine the core set of Content Providers to test. We additionally collected Content Provider URIs that we discovered during other phases of testing. For each URI, we attempted to execute each type of database operation without any permissions. If a security exception was thrown, we recorded the required permission. We added and tested combinations of permissions to identify multiple or substitutable permission requirements. Each Content Provider was tested until security exceptions were no longer thrown for a given operation, indicating the minimum set of permissions required to complete that operation.

## Intents

We built a pair of applications to send and receive Intents. The Android documentation does not provide a single, comprehensive list of the available system Intents, so we scraped the public API to find string constants that could be the contents of an Intent.<sup>1</sup> We sent and received Intents with these constants between our applications. We also triggered system Intents by sending and receiving text messages, sending and receiving phone calls, connecting and disconnecting WiFi, and connecting and disconnecting Bluetooth devices. For all of these tests, we recorded whether permission checks occurred and whether the Intents were delivered or received successfully.

# 3.4 Permission Map Results

Our testing of the Android application platform resulted in a permission map that correlates permission requirements with API calls, Content Providers, and Intents. In this section, we discuss our coverage of the API, compare our results to the official Android documentation, and present characteristics of the Android API and permission map.

## Coverage

The Android API consists of 1,665 classes with a total of 16,732 public and private methods. We attained **85%** coverage of the Android API through two phases of testing. (We define a method as *covered* if we executed it without generating an exception; we do not measure branch coverage.) Randoop attained an initial method coverage of 60%, spread across all packages. We supplemented Randoop's coverage with our proprietary test generation tool, accomplishing close to 100% coverage of the methods that belong to classes with at least one permission check.

<sup>&</sup>lt;sup>1</sup>For those familiar with Android terminology, we searched for Intent *action* strings.

The uncovered portion of the API is due to native calls and the omission of secondphase tests for packages that did not yield permission checks in the first phase. First, native methods often crashed the application when incorrect parameters were supplied, making them difficult to test. Many native method parameters are integers that represent pointers to objects in the native code, making it difficult to supply correct parameters. Approximately one-third of uncovered methods are native calls. Second, we decided to omit supplemental tests for packages that did not reveal permission checks during the Randoop testing phase. If Randoop did not trigger at least one permission check in a package, we did not add more tests to the classes in the package.

## **Comparison With Documentation**

Clear and well-developed documentation promotes correct permission usage and safe programming practices. Errors and omissions in the documentation can lead to incorrect developer assumptions and overprivilege. Android's documentation of permissions is limited, which is likely due to their lack of a centralized access control policy. Our testing identified 1207 API calls with permission checks. We compare this to the Android 2.2 documentation.

We crawled the Android 2.2 documentation and found that it specifies permission requirements for 78 methods. The documentation additionally lists permissions in several class descriptions, but it is not clear which methods of the classes require the stated permissions. Of the 78 permission-protected API calls in the documentation, our testing indicates that the documentation for 6 is incorrect. It is unknown to us whether the documentation or implementation is wrong; if the documentation is correct, then these discrepancies may be security errors.

Three of the method documentation errors list an incorrect permission in place of another. One error documents an API call as being protected by the Dangerous permission MANAGE\_ACCOUNTS, when it can actually be accessed with the lower-privilege Normal permission GET\_ACCOUNTS. Another API call is described as requiring the ACCESS\_COARSE\_UPDATES permission, which does not exist. Due to this documentation error, 5 of the 900 applications that we study in §3.6 request this permission. The third API call description lists a BLUETOOTH permission requirement, when the method is in fact protected with BLUETOOTH\_ ADMIN.

The other three documentation errors pertain to methods with multiple permission requirements. One method is described as requiring one permission, but two are required for the method to be correctly invoked. The other two methods are documented as requiring one permission, but two are accepted (i.e., either permission will suffice).

## **Characterizing Permissions**

Based on our permission map, we characterize how permission checks are distributed throughout the API.

#### CHAPTER 3. ANALYZING ANDROID PERMISSIONS

| Permission             | Usage |
|------------------------|-------|
| BLUETOOTH              | 85    |
| BLUETOOTH_ADMIN        | 45    |
| READ_CONTACTS          | 38    |
| ACCESS_NETWORK_STATE   | 24    |
| WAKE_LOCK              | 24    |
| ACCESS_FINE_LOCATION   | 22    |
| WRITE_SETTINGS         | 21    |
| MODIFY_AUDIO_SETTINGS  | 21    |
| ACCESS_COARSE_LOCATION | 18    |
| CHANGE_WIFI_STATE      | 16    |

Table 3.1: The 10 most commonly checked permissions within the Android API.

## API Calls

We examined the Android API to see how many methods and classes have permission checks. We present the number of permission checks, unused permissions, hierarchical permissions, permission granularity, and class characteristics.

Number of Permissions Checks. We identified 1207 API calls with permission checks, which is 6.26% of all API methods (including hidden and private methods). Of those, 779 are in normal API classes, and 428 are in proxy interfaces that communicate with system services. Table 3.1 provides the rates of the most commonly-checked permissions for the normal API (excluding interfaces, many of which duplicate functionality provided in the normal API).

Unused Permissions. We found that some permissions are defined by the platform but never used within the API. For example, the BRICK permission is never used, despite being oft-cited as an example of a particularly dire permission [87]. The only use of the BRICK permission is in dead code that is incapable of causing harm to the device. Our testing found that 20 of the 134 Android-defined permissions are unused. For each case where a permission was never found during testing, we searched the source tree to verify that the permission is not used. We also searched for online code snippets that include the unused permissions and found that one is used by a phone manufacturer that added a custom 4G class to the API.

Hierarchical Permissions. The names of many permissions imply that there are hierarchical relationships between them. Intuitively, we expect that more powerful permissions should be substitutable for lesser permissions relating to the same resource. However, we find no evidence of planned hierarchy. Our testing indicates that BLUETOOTH\_ADMIN is not substitutable for BLUETOOTH, nor is WRITE\_CONTACTS substitutable for READ\_CONTACTS. Similarly, CHANGE\_WIFI\_STATE cannot be used in place of ACCESS\_WIFI\_STATE.

Only one pair of permissions has a hierarchical relationship: ACCESS\_COARSE\_LOCATION and ACCESS\_FINE\_LOCATION. Every method that accepts the COARSE permission also ac-

cepts FINE as a substitute. We found only one exception to this, which may be a bug: TelephonyManager.listen() accepts either ACCESS\_COARSE\_LOCATION or READ\_PHONE\_STATE, but it does not accept ACCESS\_FINE\_LOCATION.

**Permission Granularity.** If a single permission is applied to a diverse set of functionality, applications that request the permission for a subset of the functionality will have unnecessary access to the rest. Android aims to prevent this by splitting functionality into multiple permissions when possible, and their approach has been shown to benefit platform security [51]. As a case study, we examine the division of Bluetooth functionality, as the Bluetooth permissions are the most heavily checked permissions.

We find that the two Bluetooth permissions are applied to 6 large classes. They are divided between methods that change state (BLUETOOTH\_ADMIN) and methods that get device information (BLUETOOTH). The BluetoothAdapter class is one of several that use the Bluetooth permissions, and it appropriately divides most of its permission assignments. However, it features some inconsistencies. One method only returns information but requires the BLUETOOTH\_ADMIN permission, and another method changes state but requires both permissions. This type of inconsistency may lead to developer confusion about which permissions are required for which types of operations.

**Class Characteristics.** Figure 3.2 presents the percentage of methods that are protected per class. We initially expected that the distribution would be bimodal, with most classes protected entirely or not at all. Instead, however, we see a wide array of class protection rates. Of these classes, only 8 require permissions to instantiate an object, and 4 require permissions only for the object constructor.

#### **Content Providers and Intents**

We examined Content Providers to determine whether they are protected by permissions. We investigated a total of 62 Content Providers. We found that there are 18 Content Providers that do not have permissions for any of the methods that we tested (insert, query, update, and delete). All of the Content Providers that lack permissions are associated with the content://media content URI.

We examined Intent communication and measured whether permissions are required for sending and receiving Intents. When sending broadcast Intents, 40 broadcasts are prohibited by non-system senders, 6 require permissions before sending the Intent, and 2 can be broadcast but not received by system receivers. When receiving broadcast Intents, 5 receiver types are required to have permissions. When sending Intents to start Activities, 4 Intent messages require permissions. When starting Services, 2 Intents require permissions.

# 3.5 Application Analysis Tool

We built a static analysis tool, Stowaway, which analyzes an Android application and determines the maximum set of permissions it may require. Stowaway analyzes the application's



Figure 3.2: A histogram of the number of classes, sorted by the percentage of the classes' methods that require permissions. The numbers shown represent ranges, i.e., 10% represents [10 - 20%). We only consider classes with at least 1 permission check.

use of API calls, Content Providers, and Intents and then uses the permission map built in §3.3 to determine what permissions those operations require.

Compiled applications for the Android platform include Dalvik executable (DEX) files that run on Android's Dalvik Virtual Machine. We disassemble application DEX files using the publicly available Dedexer tool [79]. Each stage of Stowaway takes the disassembled DEX as input.

# API Calls

Stowaway first parses the DEX files and identifies all calls to standard API methods. Stowaway tracks application-defined classes that inherit methods from Android classes so we can differentiate between application-defined methods and Android-defined inherited methods. We use heuristics to handle Java reflection and two unusual permissions.

**Reflection.** Java reflection is a challenging problem [30, 67, 81]. In Java, methods can be reflectively invoked with java.lang.reflect.Method.invoke() or java.lang.reflect. Constructor.newInstance(). Stowaway tracks which Class objects and method names are propagated to the reflective invocation. It performs flow-sensitive, intra-procedural static analysis, augmented with inter-procedural analysis to a depth of 1 method call. Within each method body, it tracks the value of each String, StringBuilder, Class, Method, Constructor, Field, and Object. We also track the state of static member variables of these types. We identify method calls that convert strings and objects to type Class, as well as method calls that convert Class objects to Methods, Constructors, and Fields.

We also apply Android-specific heuristics to resolving reflection by handling methods and fields that may affect reflective calls. We cannot model the behavior of the entire Android and Java APIs, but we identify special cases. First, Context.getSystemService(String) returns different types of objects depending on the argument. We maintain a mapping of arguments to the types of return objects. Second, some API classes contain private member variables that hold references to hidden interfaces. Applications can only access these member variables reflectively, which obscures their type information. We created a mapping between member variables and their types and propagate the type data accordingly. If an application subsequently accesses methods on a member variable after retrieving it, we can resolve the member variable's type.

**Internet.** Any application that includes a *WebView* must have the Internet permission. A WebView is a user interface component that allows an application to embed a web site into its UI. WebViews can be instantiated programmatically or declared in XML files. Stowaway identifies programmatic instantiations of WebViews. It also decompiles application XML files and parses them to detect WebView declarations.

**External Storage.** If an application wants to access files stored on the SD card, it must have the WRITE\_EXTERNAL\_STORAGE permission. This permission does not appear in our permission map because (1) it is enforced entirely using Linux permissions and (2) can be associated with any file operation or API call that accesses the SD card from within the library. We handle this permission by searching the application's string literals and XML files for strings that contain sdcard; if any are found, we assume WRITE\_EXTERNAL\_STORAGE is needed. Additionally, we assume this permission is needed if we see API calls that return paths to the SD card, such as Environment.getExternalStorageDirectory().

## **Content Providers**

Content Providers are accessed by performing a database operation on a URI. Stowaway collects all strings that could be used as Content Provider URIs and links those strings to the Content Providers' permission requirements. Content Provider URIs can be obtained in two ways:

- A string or set of strings can be passed into a method that returns a URI. For example, the API call android.net.Uri.parse("content://browser/bookmarks") returns a URI for accessing the Browser bookmarks. To handle this case, Stowaway finds all string literals that begin with content://.
- 2. The API provides Content Provider helper classes that include public URI constants. For example, the value of android.provider.Browser.BOOKMARKS\_URI is content: //browser/bookmarks. Stowaway identifies the use of known URI constants, and we created a mapping from all known URI constants to their string values.

A limitation of our tool is that we cannot tell which database operations an application performs with a URI; there are many ways to perform an operation on a Content Provider, and users can set their own query strings. To account for this, we say that an application may require any permission associated with any operation on a given Content Provider URI. This provides an upper bound on the permissions that could be required in order to use a specific Content Provider.

#### Intents

We use ComDroid [37] to detect the sending and receiving of Intents that require permissions. ComDroid performs flow-sensitive, intra-procedural static analysis, augmented with limited inter-procedural analysis that follows method invocations to a depth of one method call. ComDroid tracks the state of Intents, registers, sinks (e.g., sendBroadcast), and application components. When an Intent object is instantiated, passed as a method parameter, or obtained as a return value, ComDroid tracks all changes to it from its source to its sink and outputs all information about the Intent and all components expecting to receive messages.

Stowaway takes ComDroid's output and, for each sent Intent, checks whether a permission is required to send that Intent. For each Intent that an application is registered to receive, Stowaway checks whether a permission is required to receive the Intent. Occasionally ComDroid is unable to identify the message or sink of an Intent. To mitigate these cases, Stowaway searches for protected Intents in the list of all string literals in the application.

# 3.6 Application Analysis Results

We applied Stowaway to 940 Android applications to identify the prevalence of overprivilege. Stowaway calculates the maximum set of Android permissions that an application may need. We compare that set to the permissions actually requested by the application. If the application requests more permissions, then it is overprivileged.

Our full set of applications consists of 964 Android 2.2 applications.<sup>2</sup> We set aside 24 randomly selected applications for tool training, leaving 940 for analysis.

## Manual Analysis

#### Methodology

We randomly selected 40 applications from the set of 940 and ran Stowaway on them. Stowaway identified 18 applications as overprivileged. We then manually analyzed each overprivilege warning to attribute it to either tool error (i.e., a false positive) or developer error. We looked for three types of false positives:

1. Stowaway misses an API, Content Provider, or Intent operation that needs a permission. For example, reflective API calls are potential failure points.

 $<sup>^{2}</sup>$ In October 2010, we downloaded the 100 most popular paid applications, the 764 most popular free applications, and 100 recently added free applications from the Android Market.

- 2. Stowaway correctly identifies the API, Content Provider, or Intent operation, but our permission map lacks an entry for that platform resource.
- 3. The application sends an Intent to some other application, and the recipient accepts Intents only from senders with a certain permission. Stowaway cannot detect this case because we cannot determine the permission requirements of other non-system applications.

We reviewed the 18 applications' bytecode, searching for any of these three types of error. If we found functionality that could plausibly pertain to a permission that Stowaway identified as unnecessary, we manually wrote additional test cases to confirm the accuracy of our permission map. We investigated the third type of error by checking whether the application sends Intents to pre-installed or well-known applications. When we determined that a warning was not a false positive, we attempted to identify why the developer had added the unnecessary permission.

We also analyzed overprivilege warnings by running the application in our modified version of Android (which records permission checks as they occur) and interacting with it. It was not possible to test all applications at runtime; for example, some applications rely on server-side resources that have moved or changed since we downloaded them. We were able to test 10 of the 18 application in this way. In each case, runtime testing confirmed the results of our code review.

#### **False Positives**

Stowaway identified 18 of the 40 applications (45%) as having 42 unnecessary permissions. Our manual review found that 17 applications (42.5%) were overprivileged, with a total of 39 unnecessary permissions. This represents a **7%** false positive rate.

All three of the false warnings were caused by incompleteness in our permission map. Each was a special case that we failed to anticipate. Two of the three false positives were caused by applications using Runtime.exec to execute a permission-protected shell command. (For example, the logcat command performs a READ\_LOGS permission check.) The third false positive was caused by an application that embeds a web site that uses HTML5 geolocation, which requires a location permission. We wrote test cases for these scenarios and updated our permission map.

Of the 40 applications in this set, 13 contain at least one reflective call that our static analysis tool cannot resolve. 8 of them are overprivileged. This means that applications with at least one unresolved reflective call are overprivileged at a rate of 61%, and other applications are overprivileged at a rate of 33%. However, we investigated the unresolved reflective calls and do not believe they led to false positives.

#### **Developer Error**

In some cases, we were able to determine why developers asked for unnecessary permissions.

**Permission Name.** Developers sometimes request permissions with names that sound related to their applications' functionality, even if the permissions are not required. We saw several instances of this. One application registers to receive the android.net.wifi.STATE\_ CHANGE Intent and unnecessarily requests the ACCESS\_WIFI\_STATE permission. Another unnecessarily requests the MOUNT\_UNMOUNT\_FILESYSTEMS permission to send and receive the android.intent.action.MEDIA\_MOUNTED Intent. Despite their names, neither permission is actually required.

**Related Methods.** Some classes contain a mix of permission-protected and unprotected methods. We have observed applications using unprotected methods but requesting permissions that are required for other methods in the same class. For example, android. provider.Settings.Secure includes both setters and getters. The setters require the WRITE\_SETTINGS permission, and the getters do not. Two applications use the getters and not the setters, but request the WRITE\_SETTINGS permission.

Copy and Paste. Popular message boards contain Android code snippets and advice about permission requirements. Sometimes this information is inaccurate, and developers who copy it will overprivilege their applications. For example, one application in our data set registers to receive the android.net.wifi.STATE\_CHANGE Intent and requests the ACCESS\_ WIFI\_STATE permission. As of May 2011, the third-highest Google search result for that Intent contains the incorrect assertion that it requires that permission [85].

**Deputies.** An application can send an Intent to another *deputy* application, asking the deputy to perform an operation. If the deputy makes a permission-protected API call, then the deputy needs the permission. The sender of the Intent, however, does not. We noticed instances of applications requesting permissions for actions that they asked deputies to do. For example, one application asks the Android Market to install another application. The sender asks for INSTALL\_PACKAGES, which it does not need because the Market application does the installation. Another application asks the built-in camera application to take photos, yet requests the CAMERA permission for itself.

**Testing Artifacts.** A developer might add a permission during testing and then forget to remove it when the test code is removed. For example, ACCESS\_MOCK\_LOCATION is typically used only for testing but can be seen in released applications.

Confusion over permission names, related methods, and Intents could be addressed with improved API documentation. We recommend listing permission requirements on a permethod (rather than per-class) basis. Confusion over deputies could be reduced by clarifying the relationship between permissions and pre-installed system applications.

We were not always able to determine the reason for overprivilege, and there are two other potential causes of overprivilege that we did not investigate. First, permissions that are unnecessary in Android 2.2 could be necessary in older Android releases. Older versions of the API might include permission checks that are no longer present in Android 2.2. Old or backwards-compatible applications therefore might have seemingly extra permissions. Second, developers are incentivized to ask for unnecessary permissions because applications will not receive automatic updates if the updated version of the application requests more permissions [51].

#### CHAPTER 3. ANALYZING ANDROID PERMISSIONS

| Permission             | Usage |
|------------------------|-------|
| ACCESS_NETWORK_STATE   | 15%   |
| READ_PHONE_STATE       | 15%   |
| ACCESS_WIFI_STATE      | 8%    |
| WRITE_EXTERNAL_STORAGE | 7%    |
| ACCESS_MOCK_LOCATION   | 6%    |
| CALL_PHONE             | 6%    |
| ACCESS_COARSE_LOCATION | 5%    |
| CAMERA                 | 5%    |
| INTERNET               | 5%    |
| WAKE_LOCK              | 5%    |

Table 3.2: The 10 most common unnecessary permissions, and the percentage of overprivileged applications that request them.

## Automated Analysis

We ran Stowaway on 900 Android applications. Overall, Stowaway identified 323 applications (35.8%) as having unnecessary permissions. Stowaway was unable to resolve some applications' reflective calls, which might lead to a higher false positive rate in those applications. Consequently, we discuss applications with unresolved reflective calls separately from other applications.

#### **Fully Resolved Applications**

Stowaway was able to completely resolve<sup>3</sup> all reflective calls for 671 applications, meaning that it should have identified all API access for those applications. Stowaway produces overprivilege warnings for 31% of the 671 applications. Most are overprivileged by a small number of permissions: 56% of overprivileged applications have 1 extra permission, and 95% have 4 or fewer extra permissions. Although a third of applications are overprivileged, the low degree of per-application overprivilege indicates that developers are attempting to add correct permissions rather than arbitrarily requesting large numbers of unneeded permissions.

Table 3.2 shows the 10 most common unnecessary permissions among the overprivileged applications whose reflective calls were completely resolved. We hypothesized that the unnecessary use of these permissions could be partially explained by some of the same factors that we observed in the manual review:

• *Permission Names.* The ACCESS\_NETWORK\_STATE and ACCESS\_WIFI\_STATE permissions have similar-sounding names, although they are required by different classes. It appears that developers often request them in pairs, even if only one is necessary. Of

 $<sup>^3 \</sup>rm We$  refer to a reflective call as *completely resolved* if we match at least one API call to its invocation or instantiation.

the applications that unnecessarily request the network permission, 45% legitimately require the WiFi permission. Of the applications that unnecessarily request the WiFi permission, 62% legitimately need the network permission.

- *Testing Artifacts.* All of the applications that include the ACCESS\_MOCK\_LOCATION permission likely left it in from testing. 100% of those applications also include a real location permission.
- Deputies. Of the applications that unnecessarily request the CAMERA permission, 80% send an Intent that opens the Camera application to take a picture. 90% of the applications that unnecessarily request INTERNET send an Intent that opens the browser. Of the applications that unnecessarily request CALL\_PHONE, a third send an Intent to the default Phone Dialer application.

In addition to the 10 most common unnecessary permissions, 7% of all overprivileged applications request ACCESS\_GPS or ACCESS\_LOCATION. These permissions are from an early version of Android and have not existed since 2008. The permissions may be required for backwards-compatibility with out-of-date phones. They alternately could have been added by confused developers because code snippets with the permissions still appear on message boards.

We additionally find that 9% of overprivilege applications request unneeded Signature/System permissions. Standard versions of Android will silently refuse to grant those permissions to these applications. The permissions were either requested completely in error, or the developers removed the related code after discovering it did not work.

We believe that Stowaway should produce approximately the same false positive rate for these applications as it did for the set of 40 that we evaluated in §3.6, i.e., 7%. Applications could also be *more* overprivileged in practice than indicated by our tool, due to dead or unreachable code.

#### Applications With Unresolved Reflective Calls

For 229 of the 900 applications (25%), Stowaway was unable to resolve the target of one or more reflective calls. Stowaway identifies 50% of the 229 applications as overprivileged. This overprivilege warning rate is significantly different from the 31% rate for applications without unresolved reflective calls.

We do not know what causes the difference between the overprivilege warning rates, but we can hypothesize two possible causes. First, Stowaway might have a higher false positive rate in applications with unresolved reflective calls. If an application's unresolved reflective calls access permission-protected resources, Stowaway will erroneously report that the permission is unused. Second, applications that use Java reflection in complicated ways might have a higher rate of actual overprivilege due to a correlated trait such as third-party library usage. We suspect that both factors play a role in the higher overprivilege warning rate. Although our manual review (§3.6) did not find any applications whose unresolved reflective calls led to overprivilege, a subsequent review of additional applications identified some erroneous warnings that were caused by unresolved reflection. However, we do not feel that false positives completely account for the increased warning rate for unresolved applications. Our manual review of 40 randomly chosen applications yielded similarly skewed overprivilege rates despite a lack of reflection-induced false positives.

Overall, Stowaway is able to resolve 58% of reflective calls in the 900 applications. Reflective calls comprise less than 1% of all API calls made by the 900 applications, although 60% of all applications make at least one reflective call. Stowaway fails on complicated reflection use, like the creation of method names based on environment variables, direct generation of Dalvik bytecode, and storing Method and Class objects in hash tables. It is an interesting open problem to improve the resolution of reflective calls in Android applications.

# 3.7 Related Work

Android Permissions. Previous studies of Android applications have been limited in their understanding of permission usage. Enck et al. apply Fortify's Java static analysis tool to decompiled applications; they analyze a large set of applications and study their API use [45]. However, they are limited to studying applications' use of a small number of permissions and API calls. In a recent study, Felt et al. manually classify a small set of Android applications as overprivileged or not [51]. They were unable to reliably differentiate between necessary and unnecessary permissions because of limited Android documentation. Kirin [46] reads application permission requirements during installation and checks them against a set of security rules that define potentially dangerous combinations of permissions. They rely solely on developer permission requests, rather than examining whether or how permissions are used by applications. Barrera et al. examine the granularity of the Android permission system from the perspective of applications [26]. They analyze 1,100 Android applications' permission requirements and use self-organizing maps to visualize which permissions are used in applications with similar characteristics. This work relies on the permissions requested by the applications. It does not examine whether the permissions are used; as our work shows, correlations between requested permissions may be due to developer error rather than related usage. Our mapping of permission requirements can be used to greatly increase the scope of application API and permission analysis.

Vidas et al. [89] provide a tool that performs an overprivilege analysis on application source code. Their tool could be improved by using our permission map; theirs is based on the limited Android documentation. Our static analysis tool also performs a more sophisticated application analysis. Unlike their Eclipse plugin, Stowaway attempts to handle reflective calls, Content Providers, and Intents.

In concurrent work, Gibler et al. [54] applied static analysis to the Android API to find permission checks. Their permission map includes internal methods within the system
process that are not reachable across the RPC boundary, which we excluded because applications cannot access them. Unlike our dynamic approach, their static analysis might have false positives, will miss permission checks in native code, and will miss Android-specific control flow.

**Java Testing.** Randoop is not the only Java unit test generation tool. Eclat [77] is a predecessor to Randoop that also uses a feedback-directed approach but requires an example execution as input. Given the size of the Android API, building such an example execution would be a challenge. Enhanced JUnit [49] generates tests by chaining constructors to some fixed depth. However, it does not use subtyping to provide instances and relies on bytecode as input. JCrasher [38] is a randomized input generator that introduces the notion of a parameter graph to drive methods' parameter space exploration. Unlike Enhanced JUnit, it looks at types and can chain methods that way. However, JCrasher requires program bytecode and an example execution trace as input. Additionally, JCrasher focuses on error discovery and not operational testing like Randoop. Palulu [23] combines dynamic call sequence graph inference with random testing. Like many of the other tools, it requires example executions to build a model of method sequences to guide random test input generation. Korat [32] requires formal specifications of methods as input and uses the method precondition to generate test cases and the method postcondition to check the correctness of the output. As this requires formal specification for each method, this approach is infeasible for post-facto testing of the Android API.

Java Reflection. Handling Java reflection is necessary to develop sound and complete program analyses. However, resolving reflective calls is an area of open research. Livshits et al. created a static algorithm which approximates reflective targets by tracking string constants passed to reflections [67]. Their approach falls short when the reflective call depends on user input or environment variables. We use the same approach and suffer from the same limitations. They improve their results with developer annotations, which is not a feasible approach for our domain. A more advanced technique combines static analysis with information about the environment of the Java program in order to resolve reflections [81]. However, their results are sound only if the program is executed in an identical environment as the original evaluation. Even with their modifications, they are able to resolve only 74% of reflective calls in the Java 1.4 API. We do not claim to improve the state of the art in resolving Java reflection; instead, we focus on domain-specific heuristics for how reflection is used in Android applications.

## 3.8 Conclusion

In this chapter, we developed tools to detect overprivilege in Android applications. We applied automated testing techniques to Android 2.2 to determine the permissions required to invoke each API method. We developed a tool, Stowaway, that generates the maximum set of permissions needed for an application and compares them to the set of permissions

actually requested. We applied Stowaway to 940 Android applications and found that about one-third of them are overprivileged. Finally, we showed that applications generally are overprivileged by only a few permissions, indicating that developers attempt to obtain least privilege for their applications but fall short due to API documentation errors and lack of developer understanding.

## 3.9 Acknowledgments

This work appeared in part in CCS 2011 [50]. We thank our collaborators: Adrienne Felt for her work on the customizable test case generation tool, platform modifications, and testing, and Steve Hanna for his work on verifying content provider permissions.

## Chapter 4

## Analyzing Web Interaction in Android

## 4.1 Introduction

Mobile devices and platforms are a rapidly expanding, divergent marketplace. Application developers are forced to contend with a multitude of Android mobile phones and tablets; customized OS branches (e.g., Kindle Fire, Nook Tablet); and a score of competing platforms including iOS and Windows Phone. Android developers are responding to the challenge of supporting multiple platforms through the use of WebViews, which allow HTML content to be displayed within an application. At a high level, WebViews provide the same functionality as a web browser, but allow full customizability with respect to how and what content is displayed (e.g., navigation UIs, full screen, etc). These in-application browsers allow developers to write code in platform-neutral HTML and JavaScript that can be displayed by any device and version. Furthermore, application updates become simple. Developers merely update the HTML content downloaded by an application.

While this rich interaction simplifies developer support for multiple platforms, it exposes applications to attack. In this chapter, we explore two WebView vulnerabilities: *excess authorization*, where malicious JavaScript can invoke Android application code, and *filebased cross-zone scripting*, which exposes a device's file system to an attacker.

We build a tool, Bifocals, that analyzes Android applications reliant on WebViews to detect these vulnerabilities. Then, we use Bifocals to characterize the prevalence of vulnerable code. We found 67 applications with WebView-related vulnerabilities (11% of applications containing WebViews). Based on our findings, we suggest a modification to WebView security policies that would protect over 60% of the vulnerable applications with little burden on developers.

We make the following contributions:

- We build a tool to identify vulnerable WebViews.
- We measure the prevalence and impact of vulnerable WebViews.
- We suggest solutions to mitigate these vulnerabilities and we evaluate those solutions.

```
API call
setWebViewClient(WebViewClient client)
addJavascriptInterface(Object object,
String name)
getSettings().setJavaScriptEnabled(...)
```

Table 4.1: Select list of API calls used to customize WebView behavior

## 4.2 Application and Web Interaction

To understand vulnerabilities in WebViews, we must first understand the features provided by WebViews. The WebView class allows developers to display data from web pages and files within the confines of the application, seamlessly integrating web content and application content. Through the WebView, not only can developers set the content to be displayed, but they can also specify the layout and behavior of the WebView. They can choose to display the address bar, track the browsing history, control the zoom, allow searches, etc. Essentially, the WebView class allows a developer to create their own custom, embedded web browser.

Alternatively, web content can be displayed by sending a request to a browser application on the mobile device to load the content. We will focus on the WebView approach to displaying web content as opposed to the browser approach. Customizations in a WebView can lead to security problems, while browsers are separate applications outside of an application's security boundary.<sup>1</sup>

We discuss how WebViews are created and how they can be customized in more detail.

#### WebView API

The WebView API allows developers to display content in various formats. WebViews can load (1) web content using the HTTP or HTTPS protocols, (2) files from the file system via "file://," and (3) HTML via "data://." By default, a basic WebView does not execute JavaScript nor can the web content interact with the application in any way. If the user clicks on a link within the WebView, the application is exited and the subsequent URI is loaded separately by the device's default web browser.

#### WebView Customizations

We discuss relevant WebView customizations that can be made by the developer. We list the APIs in Table 4.1.

<sup>&</sup>lt;sup>1</sup>We use the term "web browser" to specifically reference a device's default web browsing application and "WebView" to refer to developer customized views.

WebSettings (Javascript and file access). Each WebView contains its own WebSetting. The Android WebSettings class manages the settings of a WebView:

- Javascript execution in a webpage can be enabled by calling setJavaScriptEnabled() on the WebSetting. By default, JavaScript execution is off.
- Access to the local file system (e.g. loading a file in a WebView) is enabled by calling setAllowFileAccess(). By default, WebViews have file system access.<sup>2</sup>
- Access to files by JavaScript running in the context of a file scheme URI is enabled by calling setAllowFileAccessFromFileURLs(). By default, WebViews grant this access for API versions prior to Jelly Bean.
- Access to content from any origin by JavaScript running in the context of a file scheme URI is enabled by calling setAllowUniversalAccessFromFileURLs(). By default, WebViews grant this access for API versions prior to Jelly Bean.

WebViewClient (Navigation ability). A WebView may or may not have an associated WebViewClient. The Android WebViewClient class is an event handler that allows developers to specify how content is rendered. By subclassing this client, the developer can specify what actions should be taken when the page finishes loading, a resource is loaded, an error is received, etc. Most notably, it allows the developer to specify the navigation behavior of the WebView (i.e., what action should be taken when the user clicks on a link in the WebView.) By overriding the default shouldOverrideUrlLoading(WebView view, String url) method, the developer can modify the URI or take different actions based on the contents of the URI. For example, a developer may specify that the URI be loaded in the WebView if it is on a specific domain, otherwise it launches the URI via web browser.

The default behavior of the WebView when the user clicks on a link in the WebView depends on the WebViewClient. We show this in Table 4.2. A WebView without a WebViewClient launches the web browser. If the WebView has a WebViewClient, the behavior depends on the shouldOverrideUrlLoading() method. If the method is not overridden or it returns false, then URIs are launched in the WebView. Otherwise, the behavior depends on the implementation of the method.

Interfaces (Code access). Developers can also give web content access to the application's internal Java code. By calling addJavascriptInterface(Object object, String name), the developer provides a handle to an application's interface to be used by JavaScript in loaded pages. For example:

<sup>&</sup>lt;sup>2</sup>Regardless, access to an application's assets and resources (located at file:///android\_asset and file:///android\_res) is always granted within each application.

| Has WVC? | shouldOverride()? | Loaded in:       |
|----------|-------------------|------------------|
| No       | N/A               | Browser          |
| Yes      | Default           | WebView          |
|          | Returns false     | WebView          |
|          | Returns true      | Depends on impl. |

Table 4.2: How navigation events are handled, based on properties of the WebViewClient (WVC)

```
WebView wv = new WebView();
wv.getSettings().setJavaScriptEnabled(true);
wv.addJavascriptInterface(new MyClass(), "mycls");
wv.loadURL("http://www.foo.com");
```

The above code creates a WebView where its web contents can invoke methods in MyClass. Assuming MyClass contains public functions test1() and test2(), any webpage loaded in the WebView can invoke the methods with this JavaScript:

```
<script>
mycls.test1();
mycls.test2();
</script>
```

WebViews provide a way to meld applications with web content. Developers can allow JavaScript to invoke registered application methods, potentially enabling application state to be altered on the fly; and they control how a user may navigate pages. These can be powerful mechanisms towards providing a rich, interactive user experience. However, they can also introduce security vulnerabilities. We discuss WebView attacks in the next section.

## 4.3 Attacks

The use of WebViews exposes applications to a larger attack surface. We discuss two main types of vulnerabilities we identified: excess authorization and file-based cross-zone scripting, and the relevant threat model for attackers to exploit these vulnerabilities.

## Threat Model

We assume developers are not malicious, though they may have varying levels of expertise in developing on the Android platform. While the application itself is trusted, the open network it passes data over should not be. Likewise, the web content it could access also should not be trusted. While we assume that the initial content loaded by an application is intended and trusted, users may be exposed to third-party content in many other ways. We will discuss this in greater detail as we explain each vulnerability.

#### **Excess Authorization**

When a developer enables JavaScript execution and registers interfaces to a WebView, JavaScript content in the WebView can invoke the registered interfaces. If malicious thirdparty JavaScript gets loaded in the page, then it too can invoke the application's registered Java code. As authorization is actually granted to more web content than intended, we call this an excess authorization vulnerability. This general attack was discovered by Luo [70]. We design and conduct a large-scale measurement study to understand the prevalence of this vulnerability.

#### Repercussions

Access to the application's Java code can lead to a variety of security implications depending on the functionality of the Java code. Information leakage can occur if the invoked methods return information. Information injection can occur if the method receives data from the web. If unexpected input is passed to the Java method and the Java code is not written carefully, an attacker might be able to crash the application, corrupt data, or otherwise launch a denial of service. Privilege escalation can occur if the methods require privileges that are owned by the applications [52,55]. Malicious JavaScript, in combination with other application vulnerabilities such as inter-application messaging vulnerabilities [37], can lead to attacks on other applications installed on the device. These are just a few of the ways an attacker can wreak havoc on an application.

#### Attackers

We consider two threat models:

Malicious third-party content. There are many ways malicious JavaScript can appear in a WebView. Usually, the first-party content on the first page loaded is trusted. However, this page could also contain ads. Malicious ads containing JavaScript have appeared on popular advertising networks such as Google, Yahoo, and The New York Times [10, 11, 19]. Another way third-party content can be embedded in the page is through the use of frames. Finally, the user may navigate to third parties via links (if allowed by the WebView's settings). If any of this third-party content is malicious, it could invoke the application interfaces in ways the developer might not have anticipated.

**Network attacker.** Another variation on this vulnerability is if the device is on an insecure network. If any page or resource is loaded over an unencrypted connection (i.e., over HTTP), then a man-in-the-middle attacker could inject any page of his choosing as a response to the request and thereby inject malicious JavaScript into the WebView.

Other threats not considered in this work. Even supposedly "trusted" websites can present a threat. First, trusted parties may purposely include what they think to be benign, third-party JavaScript. Nikiforakis et al. have shown that over 88% of websites include at least one remote JavaScript library [74]. If it were malicious, it could invoke the Android application's interface.

Additionally, "trusted" websites may also contain a cross-site scripting (XSS) vulnerability that allows an attacker to load malicious JavaScript in the page [40, 48, 64]. Over 75% of web applications are estimated to be vulnerable to cross-site scripting [82]. If a page loaded in the WebView is vulnerable to persistent XSS, an attacker may be able to exploit the XSS vulnerability to introduce malicious JavaScript into the page and then attack the mobile application. If a page loaded in the WebView is vulnerable to reflected XSS and an attacker can place links in pages in the WebView that exploit the vulnerability, then the attacker can attack the mobile application (even if navigation is restricted to the trusted domain).

For the purposes of this study, we focus on malicious third-party content and network attackers. Vulnerabilities in trusted websites can be inferred by assuming that 75 - 88% of websites may also pose a threat due to remote script inclusion or XSS.

#### File-based Cross-zone Scripting

The Android WebView renderer treats everything loaded via a "file://" URL as being in the same origin. This allows any content loaded via a "file://" URL to read any file on the filesystem that the application can, including application internal storage (which is not accessible to any other application) and, if the application has permission, any file stored on the SD card. If the application loads static content via a "file://" URL, and this content includes third-party, untrusted JavaScript (or includes JavaScript over an unencrypted HTTP connection), this JavaScript gains the ability to read all the files in the filesystem that the application can.<sup>3</sup>

If the JavaScript is requested over an insecure connection, a man-in-the-middle attacker can inject malicious JavaScript. If the JavaScript is requested over HTTPS, but from an external, potentially untrusted source, the JavaScript itself could be malicious. Once malicious JavaScript is loaded, it can read files, create a network connection, and send the contents back to the attacker.

The exposed surface for this attack is admittedly smaller than the excess authorization attack. Only loaded files provide access to the vulnerability, and once the user navigates away from the "file://" scheme, the attack can no longer be launched. Similarly, the attack cannot be launched through a non-file frame. As we find in our measurement study, file-based cross-zone scripting vulnerabilities are fortunately fairly rare.

<sup>&</sup>lt;sup>3</sup>Caveat: In the latest release of Android, the Android OS was modified to require developers to explicitly enable access to "file://" URLS, reducing the opportunity for attack. For applications prior to Jelly Bean and for applications that do not set the minimum OS version to Jelly Bean, access to files is still granted by default.

## 4.4 Bifocals

We present a tool, Bifocals, which closely examines two aspects of WebView interaction, the application and the web content, to automatically identify WebView vulnerabilities in Android applications. First, we describe how we analyze Android applications to identify at-risk WebViews. Next, we describe how we crawl and analyze the web pages loaded into WebViews, to determine whether an attacker may be able to inject malicious Javascript into the WebView. Last, we describe how we put these parts together to determine the potential impact of an attack.

#### **Application Analysis**

The first step of the tool is to analyze the application to detect potential WebView vulnerabilities.

**Policy.** If a WebView enables JavaScript, registers a JavaScript interface, and loads a URI, then it may be vulnerable to an excess authorization attack (depending on the content loaded). WebViewClient settings determine whether a user can navigate away from the page while staying within the confines of the WebView. This increases the potential for attack because every page a user navigates could also contain malicious JavaScript, as opposed to just the initial landing page.

**Implementation Details.** Applications for the Android platform are comprised of Dalvik executable (DEX) files that run on Android's Dalvik Virtual Machine. We first disassemble application DEX files and extract XML content and file resources packaged with the application using the publicly available Dedexer [79] and Baksmali tools [18].

Bifocals statically analyzes the disassembled output. Static analysis is a common approach for bug finding [35,68,90]. Bifocals specifically performs flow-sensitive, interprocedural static analysis. For optimization purposes, we limit the method invocation tracking to a nesting depth of three. Experimentally, we have not seen any cases where WebView information is propagated more than three levels deep. Bifocals tracks the state of WebViews (including WebView subclasses), WebViewClients (including WebViewClient subclasses), strings, numbers, and any relevant fields, parameters, and return values.

For each method that uses WebViews (whether it passes a parameter, instantiates a WebView, or otherwise receives a WebView), Bifocals determines:

- Whether JavaScript execution has been enabled for the WebView (by detecting Web-View.getSettings().setJavaScriptEnabled(true) and identifying that the parameter value is true)
- If it allows JavaScript, what application interfaces are made accessible to the JavaScript (by tracking the interface parameter flowing into WebView.addJavascriptInterface(-Interface,...))

- 3. The URI that is being loaded (by tracking the String value that flows into the load-related API calls)
- 4. Whether a user can navigate to other webpages within the WebView (by evaluating the implementation of any methods that override WebViewClient.shouldOverride-UrlLoading())

In most cases, these properties are determined by tracking information to the WebView (string value, numbers, classes, etc.) Determining the fourth property requires a little more explanation. In addition to implicitly setting a navigation policy via the presence of the WebViewClient or using the default behavior of the WebViewClient.shouldOverrideUrl-Loading() method, developers may also apply a policy for navigation behavior through code in the WebViewClient.shouldOverrideUrlLoading() implementation. We apply a heuristic to infer navigability. If the implementation of this method returns false, then users can navigate within the WebView. If the code for this method (or any methods called within the code for this method) contains a load URI call, then users can navigate within the WebView. In the code is a message invocation to launch the web browser. In that case, the developer has set a hybrid policy (e.g., loading the page in the WebView if the domain is mysite.com and launching the browser otherwise), and we conservatively consider that any new URIs will launch the browser (limiting the navigability, and thus, the attack opportunity).

These vulnerable WebViews and the URIs loaded into them are passed to the web analysis portion of the tool.

#### Web Analysis

The second stage analyzes the URIs (websites, files, and data) that are being accessed to determine whether they might embed or navigate to third-party content.

**Policy.** For each URI, Bifocals examines the page for potentially malicious third-party content. We focus specifically on attack scenarios where malicious JavaScript may be included in the WebView via website content, insecure networks, and user navigation and not via the exploitation of XSS vulnerabilities. Although third-party content can encompass many forms of content (e.g., images, scripts, frames, etc.), we limit the definition of potentially malicious third-party content to content that can lead to the execution of untrusted script. We classify ads and frames that load third-party sites as potentially malicious. Ads can be supplied by anyone and can contain JavaScript. Similarly, frames that load external content are considered untrusted. We ignore third-party images and other content that does not contain or execute script. We also ignore non-ad-related JavaScript (e.g., non-ad <script src=...>) unless it is embedded in a third-party page. Many webpages include popular, trusted third-party JavaScript, such as Google Analytics, Facebook's "Like" button, etc., and we assume these are intentional and we do not treat them as potentially malicious.

Depending on whether a WebView is navigable, we apply the same evaluation to all pages linked from the landing page including all pages transitively reachable (up to a depth of three). Additionally, if the user can navigate to a third-party page (via links) within the WebView, we classify it as potentially malicious.

We assume that the primary website being visited and sites within the same domain are trustworthy, as well as anything belonging to the same second-level domain (the domain directly below the top-level domain in the DNS). For example, suppose a WebView loads http://mysubdomain.mysite.com. The domain mysubdomain.mysite.com, its second-level domain (SLD) mysite.com, and other subdomains of it (e.g., myothersubdomain.mysite.com) are most likely under the same jurisdiction and therefore we treat them all as trustworthy. This trust is similar to the implicit trust of cookie setting between a subdomain and its parent domain [8].

In the case of certain domains with country codes, we take the third-level domain. For example http://blogs.telegraph.co.uk's trusted domain would be telegraph.co.uk.

**Implementation Details.** To perform this analysis, we build on a basic web crawler built as a Firefox extension [61]. Given a URI, this crawler invokes Firefox, loads the page, and returns redirect information and the HTML source (including the frame source). We modified the extension to also log links on the page, frames loaded in the page, and the links within those frames.

To identify ad content, we incorporated and modified the Adblock Plus extension [1]. Adblock Plus is a browser extension that parses pages and identifies and removes ads. For every network request required to load a page, it invokes a JavaScript function shouldLoad() that returns whether that content is an ad and should be loaded. We modified Adblock Plus in two ways. First, we modified the shouldLoad() function to log the content type (e.g., script, image, document, subdocument, etc. [15]), request origin, and target location. Second, we always allow the content to be loaded but log when an ad is identified.

To simulate a mobile browser, we modify the Firefox preference file (prefs.js) to set the user-agent string to the user-agent string of an Android Browser. This way, the web behavior returned by the request is the mobile behavior, not the desktop browser behavior.

Finally, we modify URIs before loading. For URIs that load data, we prepend the HTML with data:text/html, so that the browser loads the data string as a data URI. For URIs that load data with a relative base URI, we prepend the HTML with data:text/html,<br/>base href=""">text/html, so that the browser loads the data string as a data URI. For URIs that load data with a relative base URI, we prepend the HTML with data:text/html,<br/>base href=""">text/html, so that the browser loads the data string as a data URI. For URIs that load data with a relative base URI, we prepend the HTML with data:text/html,<br/>base href=""">text/html, so that the browser renders the data and resolves all relative references.

The crawler then crawls the URIs that could be loaded into the WebView. If a vulnerability is identified or the WebView that the URI is from is not navigable, the crawling for that URI ends. Otherwise, the crawler repeats the page analysis for all links in the page and frames with the same SLD as the original URI or its redirects. We limit the crawling link depth to three for feasibility reasons.

Results from the crawler and the application analysis are then combined to identify WebViews that are fully vulnerable to the excess authorization attack. We identify file-based cross-zone scripting attacks by checking whether any of the loaded file URIs (regardless of whether interfaces are registered) contain third-party JavaScript.

#### **Impact Analysis**

There are many ways to examine the impact of a vulnerability. As discussed in Section 4.3, an attack on a WebView could result in information leakage, information injection, DoS, etc. One way to measure impact is to examine how many privileged resources an attacker would gain access to. We do this by analyzing the code invoked by the interface and determining the permissions required to execute that code.

We built a tool to determine what Android APIs a registered interface transitively grants access to (through invocation) and the permissions they correspond to. Given an interface, we analyze all methods that can be accessed in that interface (namely, all public methods and any superclasses' public methods). We assume that the attacker cannot reflectively invoke private methods, but can determine public methods via reflection or direct analysis of the target application.

For each of the directly accessible methods, we recursively analyze the methods invoked by the method and the Android API calls made in the method. If an interface method returns an object of a different class, we analyze that object's public methods as well. We apply an Android API-to-permission map [50] to determine the set of permissions used by the reachable code. To determine the permissions used by non-API calls, Android message passing, Android databases, and code invoked via Java reflection, we modify Felt et al.'s Stowaway tool [50] to identify and output the methods in which these permissions are used. If those methods are reachable, then we add the corresponding permissions to the permission set. We include both normal and dangerous permissions in the set of permissions used.

#### Limitations and discussion

**Platforms.** There are alternatives to using Firefox extensions to perform a web crawl. We could have used a command-line tool (e.g., wget), however this has limitations on the information received from the page. We preferred to use a full-featured web browser. It allowed us to leverage the existing Adblock extension, parse the loaded DOM in real-time, and fully render the content.

We chose to run this on a desktop computer with modifications to the browser preferences to spoof a mobile browser, as Firefox is more robust and efficient in crawling pages at scale. Given the massive amounts of meta-data produced from the crawl (from a large data set), performing the crawl on a mobile device would present challenges of dealing with a less robust, memory- and space-limited operating system. While it is possible for websites to rely on fields other than user-agent to determine whether it is running on a mobile device (and change content accordingly), user-agent is by far the most commonly used field. In fact, we investigated the possibility of alternate indicators (e.g., JavaScript's Navigator.platform or Navigator.appName), but we observed only the user-agent being used in the websites we crawled. Even if websites were modified based on different Navigator fields, it is more likely to change the layout, not the nature of the content (links, frames, or ads), and therefore it would not impact our results.

Ad Networks. Although we identify ads as potentially malicious, some ad networks may prohibit JavaScript from advertisers. We did not further classify ad networks based on whether a third-party advertiser could include JavaScript.

**Crawling.** One of the limitations of our crawling approach is the possibility of false negatives. Web content is dynamic. An ad or other third-party JavaScript may not always appear on a given page. To address this, we crawled each page three times.

Another potential source of false negatives is the inability to crawl all available content. We limited the crawl depth to three links, but untrusted JavaScript may be on a page that our tool did not crawl. Websites might prevent our crawler from seeing the content due to a pay-wall or login-wall. In this case, our crawler will only analyze the login page. This is not an incorrect crawl as the user would also be exposed to the page when prompted to login. We would, however, fail to crawl other content in that site. To address this, we would have to manually create dummy accounts, log in, and then crawl the page.

Finally, our crawl results may become stale. A website that includes little to no content today may have third-party content tomorrow. The web analysis only gives a snapshot of a vulnerability existing at the time the site is crawled.

Due to these limitations, our tool reports a lower bound on vulnerable applications. On the other hand, mobile applications change less frequently than web content, and we can use the number of potential WebView vulnerabilities from the application analysis to estimate an upper bound on the number of actual vulnerable WebViews.

**Static analysis.** A limitation to our static analysis approach is the risk of not deriving the correct URI. If the URI is comprised of strings that are obtained from dynamic messages (Intents), from an API call that we do not handle, or from system state (e.g., getting the device ID, getting accelerometer data), then we may fail to infer the URI loaded into the WebView. Crawling an invalid URI could result in a redirect to a different page. In most cases, we believe that the redirected page would also be representative of the content that the page would have displayed (in terms of using ads and linking to third-parties). We additionally supplement missing data by substituting logical default values for substrings that cannot be derived. For example, if the string appends a float value that we do not track, then a "1.0" is appended in its place. Our tool also does not attempt to handle implicit control flow or resolve Java reflection of the WebView API, and this could lead to false negatives. Our tool, however, does resolve Java reflection for the impact analysis (permissions), which is more likely to contain reflection. (Developers are unlikely to reflectively call the WebView API as the API is already publicly accessible.)

We considered a dynamic analysis approach to Bifocals as an alternative to our static approach. A dynamic analysis tool would be able to accurately determine dynamically set variables and state. It would also be able to confirm a vulnerability by exploiting it at run-time. However, it would be challenging to explore the full application state space to traverse all WebViews and to generate valid input for malicious JavaScript. Additionally, some Android UIs cannot be explored without user input (e.g., applications with logins). We chose a static approach because it achieves better code coverage, increasing the possibility of discovering vulnerabilities that may not have been exposed at runtime. We leave the possibility of a combined static and dynamic approach to leverage the benefits of both techniques for future work.

### 4.5 Evaluation

We ran Bifocals on 864 popular Android 2.2 applications to identify the prevalence of Web-View vulnerabilities.<sup>4</sup>

#### Characterizing the use of WebViews

**Developer use of WebViews.** We first analyzed these applications to better understand their use of WebViews. We found that 608 of the 864 applications (70.4%) contained at least one WebView in the application. Of these 608 applications, 433 (71.2% of applications with WebViews, 50.1% of all applications) contained at least one WebView in the core functionality of the application. Also, 351 applications (57.7% of applications with WebViews, 40.6% of all applications) contained at least one WebView displayed by an ad library in the application.<sup>5</sup> This suggests that use of web content in Android applications is common.

The web content displayed in a WebView can be hosted remotely or locally. We analyzed all WebViews in these applications to identify what URI is initially loaded into the WebView. In Table 4.3, we summarize the schemes used by these applications. Overall, many applications load content over HTTP or via the data scheme. Use of SSL is much less common.

**Exposure of interfaces.** We further examined how many applications allow JavaScript to invoke application code (by registering interfaces). We call these *authorized WebViews*. As indicated in Table 4.4, of the 608 applications with WebViews, we find that one-fifth of these applications have at least one authorized WebView. Furthermore, one-fifth of applications have authorized, core WebViews, while 10.8% of applications have authorized, ad

<sup>&</sup>lt;sup>4</sup>We use a dataset that consists of the 100 most popular paid applications, 764 most popular free applications, and 100 recently added free applications from the Android Market (as of Oct. 2010). After removing duplicate applications, applications that only consisted of keys to unlock paid features for free applications, and applications used for tool development and testing, we were left with a set of 864 applications for analysis.

<sup>&</sup>lt;sup>5</sup>In the rest of the section, we may shorten the phrases "WebView in the core functionality of the application" to "core WebView" or "core application" and "'WebView in an ad library in the application" to "ad WebView" or "ad application."

| Content loaded via:              | # of apps | %     |
|----------------------------------|-----------|-------|
| HTTP or HTTPS                    | 345       | 56.7% |
| http://                          | 335       | 55.1% |
| https://                         | 15        | 2.5%  |
| Local static content (file/data) | 374       | 61.5% |
| file://                          | 103       | 16.9% |
| data: (e.g., $<$ html $>$ )      | 323       | 53.1% |

Table 4.3: The types of URIs loaded into WebViews

|                       | Total | Core  | Ad    |
|-----------------------|-------|-------|-------|
| Apps with WebViews    | 608   | 433   | 352   |
| Apps with auth'ed WVs | 120   | 85    | 38    |
| %                     | 19.7% | 19.6% | 10.8% |

Table 4.4: Breakdown of applications that grant JavaScript code access by whether the WebView is in the core application or ad library

| Authorized WVs by URI scheme | # of Apps  |
|------------------------------|------------|
| http://                      | 57 (47.5%) |
| https://                     | 2(1.7%)    |
| file://                      | 19 (15.8%) |
| data:                        | 32 (26.7%) |

Table 4.5: Breakdown of authorized applications by the URI scheme used

WebViews.<sup>6</sup> This suggests that many developers do use WebView APIs to grant web content access to application content.

The 38 applications with authorized ad WebViews can be attributed to three distinct ad providers: Millennium Media [13], AdMarvel [2], and Medialets [12].

In Table 4.5, we further break these authorized WebViews down by the scheme of the URI initially loaded in the WebView. Unsurprisingly, many of these WebViews load content over the HTTP protocol, and very few use SSL. The distribution of schemes for these types of WebViews closely mirrors that for all WebViews, except that fewer of the applications loading content via data schemes expose an interface (10% vs. 16%; p = 0.025, Fisher's exact test).

Among the 85 applications that expose interfaces to core WebViews, 34 applications (40%) have WebViews where the user can navigate within the WebView, while 51 applications (60%) have WebViews that restrict navigation (by launching subsequent URLs in a

 $<sup>^{6}{\</sup>rm The}$  sum of the applications with core and ad WebViews exceed the 120 applications as some applications have both core WebViews and ad WebViews.

| Vulnerability  | Core | Ad | Total |
|----------------|------|----|-------|
| Network attack | 32   | 33 | 65    |
| Web attack     | 18   | 33 | 51    |
| Total          | 33   | 33 | 66    |

Table 4.6: The number of vulnerable apps found by Bifocals.

browser application). This is promising, as it shows that a majority of the applications have reduced their potential attack surface. However, restricting navigation does not completely eliminate the risk if the initial page incorporates third-party frames or JavaScript.

#### Automated Analysis

In summary, Bifocals found 67 applications (11.0% of applications with WebViews, and 55.8% of applications with authorized WebViews) that are vulnerable to at least one of the attacks presented. The high rate of vulnerabilities suggests that the Android WebView interface is error-prone, and the ability to expose APIs to web content is particularly risky.

#### **Excess Authentication Vulnerabilities**

We summarize the number of vulnerable applications in Table 4.6. We evaluate ad and core WebViews separately, as vulnerabilities in ad libraries can only be fixed by the ad provider, while vulnerabilities in the core application can be fixed by the application developer. Also, patching one ad library could secure multiple applications while patching vulnerabilities found in core WebViews must be done individually by each affected developer.

**Network attacker.** We found 65 applications (54.1% of applications that register interfaces) that are vulnerable to an excess authorization attack if used while connected to an insecure network. These applications load content over HTTP into an authorized WebView, and a network attacker could inject malicious Javascript into this content and invoke the APIs exposed by the application.

The impact of these vulnerabilities varies. For 18 (56.2%) of the 32 applications with this type of vulnerability in a core WebView, a network attacker gains access to API calls that use one or more Android permissions available to the application. Thus, the attacker may be able to take actions that would not be available to arbitrary web content. None of the ad libraries' WebViews give access to API calls that require permissions, so those vulnerabilities may have lower impact. It is important to note that access to permissions is only one metric to measure impact. Several other attacks may be possible even on applications whose API does not use any special permissions.

Web attacker. Bifocals found 51 applications (42.5% of applications that register interfaces) that are vulnerable to attack through malicious websites.

Many of these vulnerabilities grant a malicious website abilities that we would not expect web content to receive. 13 (72%) of the 18 applications containing a core WebView that is vulnerable to a web attack give the web attacker the ability to invoke an API that uses one or more of the application's Android permissions. In contrast, none of the ad-based vulnerabilities allow attackers to invoke code that uses permissions.

#### File-base Cross-zone Scripting Vulnerabilities

Our tool identified two applications that load files that access remote JavaScript. One of these is vulnerable to a network attack. The other requests Javascript over SSL from a trusted site, making it resistant to attack.

Upon further inspection, we find that many files loaded into a WebView are simple HTML pages with no need for JavaScript. For example, files may contain EULAs, Terms of Service, and FAQ pages.

#### Manual Analysis

We randomly selected 10 applications (of the 18 applications with a web-based excess authorization vulnerability in a core WebView) and manually analyzed these applications, in order to determine the false positive rate of Bifocals. For each selected application, we examined the code, the loaded websites, and application as installed on an Android phone. For each reported vulnerability, we confirmed that Bifocals correctly inferred the APIs registered, URIs loaded, and navigation capability of the WebView. For each loaded URI, we confirmed the crawler result: that an ad, external frame, or site was found within the navigation constraints of the WebView. We did not build an exploit.

We manually analyzed a total of 19 vulnerable WebViews across 10 applications and found no false positives. This suggests that Bifocals's false positive rate is likely below 5 - 10%.

We now discuss a few applications and the vulnerabilities we discovered.

Alive. Alive is an application that displays Japanese cartoon images. The main Activity includes a tab for a user to browse or search for other applications to install. This content is displayed in a WebView, and the landing page and linked pages contain ads. The registered interface provides code to download and install an application. This code accepts a URL as a parameter, downloads the content located at that URL, and saves it to the SD card. The expected use case is that a user can enter the tab, browse application descriptions, select an application, and click "download"; at that point, the selected application will be saved to the SD card (without any prompt to the user). After the download, the user is then prompted about whether they want to install the application. If they choose "yes," the code launches Android's application installation process.

This introduces multiple risks. One possible attack is that a network attacker or malicious advertisement could save arbitrary files to the SD card, by invoking the registered API with a URL pointing to a site controlled by the attacker. Also, an attacker could trick the user into installing a malicious application, if the attacker launches her attack when the user is browsing an application they are likely to install, or possibly through some other social engineering attack.

The Alive application has two other WebViews with a similar vulnerability. These Web-Views expose interfaces that accept a URL; when invoked, the application will download the content at the URL to the application's internal data folder instead of the SD card.

**AIM.** The AOL Instant Messaging application contains a vulnerable WebView that accesses the READ\_PHONE\_STATE permission. The application provides an interface to handle successful logins. An attacker (network or web) can use this interface to control the values of the authentication token, session key, screen name, profile URL, and icon URL. This data goes into an "IdentityPreference" data structure which gets used throughout the application, making the application vulnerable to information injection and potentially a CSRF login attack.

Ad Libraries. We also manually examined two of the three ad libraries with potential vulnerabilities: Millennial Media and AdMarvel. The third, Medialets, was obfuscated. Millennial Media is an advertising service that offers rich media ads. The registered interface allows the web content to modify the look and feel of the WebView itself, including the size and title of an overlay. AdMarvel is another advertising service that provides rich media ads. The registered interface allows web content to resize the view or change the layout settings. While neither of these libraries' interfaces invoke protected resources, an attack can still be mounted. A malicious (or benevolent) attacker can resize the WebView.

Our tool was unable to determine the URLs loaded for these WebViews (due to complexities with generating the URL string), so we manually confirmed the vulnerability and blacklisted the two libraries. It is possible that the third, obfuscated library, Medialets is also vulnerable, but we conservatively leave that out of our analysis. Only 5 applications use Medialets.

#### Evaluation of the Tool

We find that our tool is able to correctly determine the URL loaded into each WebView in most cases. In two cases, however, our tool could not fully resolve the URL. In one case, the value was set at runtime, based on the user's input into an editable text box. In the other case, the value was from device parameters. In both cases, the missing portion of the URL was a value for the URL query string. Ultimately, these query parameters did not affect the landing page, therefore the result from the crawler was correct.

In two cases, the landing page no longer existed, and in its place were squatter and GoDaddy pages, respectively. Our crawler crawled these pages and found potential vulnerabilities. We believe this to be the correct result. The squatting page would be displayed to the application user, therefore this WebView is vulnerable. In fact, this may present an even larger threat, as an attacker can easily gain access to the user's application by purchasing the domain.

#### Discussion

One limitation of our study is that our data set is two years old. Although it would be interesting to evaluate Android 4.2 applications, we feel our results would not change significantly. If anything, WebViews have increased in popularity, potentially increasing the number of applications exposed to these vulnerabilities. Furthermore, all vulnerabilities still exist in the current platform API. Only one change was made to the JavaScript interface for Android 4.2, which was to require explicit annotations to JavaScript accessible methods.<sup>7</sup> This modification is only applied to applications that set Android 4.2 as the minimum or targeted API. As of Feb. 4, 2013, only 1.4% of Android devices operate on Android 4.2 [7], and it is unlikely that many developers have set their applications to restrict distribution to the Android 4.2 platform.

## 4.6 Suggested Improvements

#### Current Shortcomings

The core of the excess authorization problem is that *any* content loaded in the WebView is able to invoke application code, making it very easy for developers to unintentionally grant untrusted sources the ability to invoke application code. We conjecture that many of the vulnerabilities we found may be attributable to developer confusion with the WebView system. In particular, we observed three significant pitfalls for developers:

- 1. WebViewClients transparently change navigation behavior. If a WebViewClient is added, the WebView is implicitly made navigable. A developer who adds a WebViewClient to alter some non-navigation feature will unwittingly make their WebView navigable, and thus may introduce an excess authorization vulnerability without realizing it.
- 2. We have observed confusion with what the shouldOverrideURLLoading() method means and does. Stack Overflow contains many questions on what the method should do [86]. Most commonly, we have observed implementations of the overridden method that load a URL and then return true. This is the equivalent of taking the default behavior and not overriding the method at all or simply returning false.
- 3. A third potential source of confusion is that developers just may not be aware that *everything* loaded in the page or navigated to can invoke the application code.

#### **Recommendations for Developers**

In light of these pitfalls with WebViews, we suggest several ways that a developer can reduce their attack exposure:

 $<sup>^7\</sup>mathrm{This}$  was announced on Feb. 14, 2013 [17].

- **Disable Javascript.** Developers can turn off JavaScript if they do not expect the web content to need it.
- **Restrict navigability.** Developers can restrict the navigability of the WebView. This restriction, however, only limits what content gets loaded via links and does not limit content within the document, such as frames or JavaScript. Consequently, it is not a complete defense.
- Limit APIs. Third, developers can restrict access to the exposed API by only registering necessary interfaces. Functionality that should not be made available to web content should be separated out into a different class.
- Use new Android mechanisms. Android just recently announced a new requirement for accessible interface methods to specifically be annotated with @Javascript-Interface for Android Jelly Bean [17]. Developers should opt in to this requirement by setting the minimum (or targeted) SDK version to Android 4.2. One caveat, however, is that while this may limit the number of methods that JavaScript can invoke and reduce accidental over-inclusion, it does nothing to prevent malicious content loaded in the page from invoking intentional interface methods. Another caveat is that this approach does not exist for devices running versions older than 4.2. Also, it may take years for the Jelly Bean version to be used by a majority of phones, and developers may not want to limit their application's user base by targeting Jelly Bean for a while.

While these approaches do not wholly prevent a vulnerability, they may limit the attack surface.

### **Recommendations for the Android Platform**

To reduce the risk of unintentional excess authorization, we recommend that the Android platform be modified so that access to an exposed interface is granted only to specified domains, instead of (as is currently the case) to all content loaded in a particular WebView. For example, if a WebView loads foo.com, only foo.com should be allowed to invoke the interface. bar.com or anything else with a different domain should not get access to the interface. Third-party web content loaded via frames should not get access to application code.

Specifically, we propose a policy that limits access by the second-level domain (SLD) of the content loaded in the WebView. The policy maintains a list of allowed SLDs for each WebView, and authorizes all content from such an SLD to invoke any interface registered with that WebView. By default, the list of allowed SLDs is initialized with the secondlevel domain of the URL initially loaded in the WebView. If this triggers a redirect, we automatically add the SLD of the target as well. This list can be supplemented by an optional developer-supplied whitelist of acceptable SLDs for each WebView (a WebViewlevel whitelist). This approach provides an automated way to secure WebViews, lowering developer burden, while providing flexibility for developers to override the policy if they intentionally want specific third-party content to access the application.

**Developer Effort.** We evaluated this approach based on the amount of developer effort that would be required to comply with it. In most cases, the developer need not do anything. We found that 97.6% of core applications that give access to code are handled automatically by our default policy and do not require any developer effort or other changes. Another 2.4% of applications would require a developer-supplied whitelist consisting of one SLD in order to maintain code access for pages redirected to a different SLD. We did not find any application in our data set that requires more than one additional SLD. Therefore, the developer burden seems to be modest.

**Effectiveness.** Our approach would patch vulnerabilities due to frames and links. It would not patch vulnerabilities due to third-party JavaScript included directly on the landing page. If the landing page contains ads or other malicious script, they would be loaded in the landing page and would gain the ability to invoke the registered interface, making the application vulnerable.

We find that of the 18 vulnerable core applications, 11 of the landing pages (61%) would be patched by our proposed policy. The remaining applications load ads directly on the landing page. Our estimate, however, may be an under-approximation of the number of patched pages. Adblock flags actual ads as well as ad providers' JavaScript (such as the Google pageads script that generates the ad). It is possible that the JavaScript subsequently loads the ad content in a frame, in which case our solution would patch the vulnerability; however, this case is not included in our count of patched applications.

Alternatives. We considered several other alternative designs: (1) where the whitelist consists of either SLDs or domains, and (2) where there is one application-wide whitelist (that applies to all WebViews in the application) or one whitelist per WebView. We found that the developer burden for all of these was modest.

We empirically found that an application-wide whitelist would not reduce the number of entries needed in a developer-supplied whitelist. Therefore, having a WebView-level whitelist would not increase developer effort, and it additionally allows the developer to set a more fine-grained access policy. This approach also allows developers to include a whitelist for third-party libraries without also granting that access for all WebViews in the application.

We also considered a whitelist of full domains, instead of SLDs; however, we found that this did not appear to provide any clear security benefit, and it slightly increases developer effort of supplying a whitelist: the number of applications that require a developer-supplied whitelist increases from 2.4% to 7.1%, and half of those applications require two entries in the whitelist, instead of one entry.

For these reasons, our proposal is to use a policy that gives access to an interface if the content is from the same SLD as the URL initially loaded in the WebView or if the content shares an SLD from an optional developer-supplied per-WebView whitelist.

## 4.7 Related Work

WebViews. We are inspired by the work of Luo et al., which identifies the potential for WebView attacks [70]. They give examples for how webpages can attack applications, how applications can attack webpages, and introduce the excess authorization vulnerability. They perform a brief, primarily manual analysis of the possibility of these vulnerabilities in applications based on the presence of WebView- and JavaScript interface registration- API calls. We extend their work by identifying variations on the basic code exposure attack and enumerating threats from different attackers, including the network attacker and attacks via remote script inclusion and XSS threats. Also, in contrast to their small-scale, manual investigation, we perform a large-scale measurement study and build an automated analysis tool to detect these vulnerabilities.

Saltzman blogged about a WebView-related attack in file-sharing applications [16]. Filesharing applications, such as DropBox or Google Drive, often save files to the application's internal file directory and can be loaded and displayed in a WebView. Assuming a malicious file gets saved, this file would then gain access to other files, potentially loading and sending them to the attacker. We present a file-based cross-zone scripting attack that is a more general form of this attack, which can occur in non-file-sharing applications. A trusted internal file, as opposed to malicious files, can load external JavaScript, giving it access to the file system.

Web Browsers. Some of our presented attack scenarios leverage vulnerabilities currently existing in the web browser. Cross-zone scripting is a category of browser attacks where the attack occurs across zone-based security boundaries. The file attack is a type of cross-zone scripting attack that takes advantage of the file origin to gain access to privileged files [16].

Cross-site scripting is another type of attack frequently examined in literature [40,48,64]. This attack is one way for malicious JavaScript to be included on a trusted site. Another way is through remote JavaScript inclusion, when a developer purposely includes third-party scripts. Nikiforakis et al. presents a large-scale measurement study on remote JavaScript inclusion [74]. They find that over 88% of websites include at least one remote JavaScript library.

## 4.8 Conclusion

While WebViews facilitate the creation of rich, interactive applications, they also introduce the potential for attack if developers are not careful. We examine vulnerabilities of WebViews and present Bifocals, which analyzes both Android applications and web content to identify vulnerabilities in applications. We discovered 67 applications that are vulnerable to attack through WebViews.

Excess authorization arises due to a mismatch in authorization expectations. A developer may intend to give code access to a specific website, but in actuality access is granted to anything loaded in the WebView. We propose changes to WebViews to grant code access based on the domain and not the WebView, thereby limiting the opportunity for exposure to malicious JavaScript. Our solution patches 60% of the vulnerabilities we found and requires very little developer effort.

# Chapter 5 Related Work

In this chapter, we briefly discuss other work on Android application vulnerabilities and program analysis tools not previously discussed in the previous chapters.

#### Analysis tools for Android

Researchers have developed analysis tools to identify other security properties in Android applications. For vulnerability detection, Davi et al. discuss privilege redelegation attacks in Android [39]. Grace et al. and Felt et al. further apply CFG-based static analysis techniques to detect these capability leaks across application boundaries [52, 55]. Following our work on message communication vulnerabilities, Lu et al. present a Dalvik static analysis tool, CHEX, that detects specific data-flow (e.g., data leakage, data injection) due to an exploitable component hijacking vulnerability [69]. CHEX can identify privilege redelegation, Intent spoofing, and other vulnerabilities. Lu et al. analyze 5, 486 applications and detect 254 potential component hijacking vulnerabilities (with a true positive rate of 81%).

Concurrent to our work on overprivilege and in contrast to our dynamic approach to deriving an API-to-permission map, Au et al., Gibler et al., and Bartel et al. build tools that take a static approach to deriving an API-to-permission map [24,25,27,54]. Our dynamic approach may fail to achieve test coverage in certain cases (e.g., when valid input variables could not be determined); therefore, it was augmented with a semi-manual verification step. The other groups' static analysis approaches allow for easier permission map generation when new versions of the Android API are released. Static analysis has limitations however. The Android API is written in both Java and C++ and method calls frequently cross process boundaries, making static analysis difficult. Static analysis may miss permissions checked in native code. Android also includes code that is only reachable under factory conditions. This means that a static approach can be prone to false positives. In contrast, our dynamic approach can detect permission checks in native code and and confirm permission checks in reachable code. For example, Bartel et al.'s approach missed permissions for 247 API calls while our dynamic approach only missed 3 API calls. They included 143 unreachable permission checks while our work had 0 unreachable permission checks.

AdDroid builds on our Stowaway tool to examine application overprivilege due to permissions only required by ad libraries [80].

Other analysis tools focus on the identification of privacy-invasive grayware or malicious applications. SCanDroid, one of the first static analysis tools for Android, takes a datacentric approach to reasoning about the consistency of security specifications concerning permissions and databases [53]. It analyzes data policies in application manifests and data flows across Content Providers. Based on its analysis, it makes a recommendation on whether an application can be installed with the permissions it has without violating the permissions of other applications. Their tool, however, requires Java source code, which generally is not available to end users.

Kim et al. present a bytecode-level static analysis tool, ScanDal, to detect privacy leaks [63]. They track location info, IDs (IMEI, IMSI, ICC-ID), audio and video eavesdroppers. PiOS similarly statically analyzes applications for privacy leaks at the binary level but targets the iOS platform [43].

Batyuk et al. and Schmidt et al. propose static analysis techniques to identify malicious Android applications [29,84]. Kirin identifies malicious applications based on security configurations in the manifest and a blacklist of undesirable permission combinations [46]. Others use dynamic approaches to identifying malware [31, 33, 92]. Crowdroid takes system call traces from real users' applications and crowdsources the detection of mobile malware [33].

In contrast to building analysis tools from scratch, Scandariato et al. apply the COTS tool, Fortify Source Code Analyzer, to open-source Android applications and use code metrics to infer the likelihood of vulnerabilities in applications [83]. Enck et al. also take advantage of Fortify SCA but avoid dataset limitations of open source applications by creating a decompiler called *ded* to generate Java source code from an Android application binary. They conduct a broad survey of vulnerabilities in Android, including standard Java and Android-specific security threats (e.g., information leakage through logs and messages, IMEI leakage, resource abuse, and ad use) [45]. Gibler et al. present AndroidLeaks, which uses static taint analysis on Android applications to identify privacy leaks [54].

#### Automatically Securing Android Applications

We are not the only ones to seek to strengthen the security and privacy of Android through platform modifications. Ongtang et al. present Saint, a modification of the Android platform for runtime enforcement of application provider policies [75]. Saint provides a means for application developers to set finer-grained policies on whom the application should trust and what it should require before interacting with other applications. Saint assumes developer knowledge in setting security policies and moves control over security decisions to the application developer.

Dietz et al. and Felt et al. present modifications to the Android platform, Quire and IPC Inspection respectively, to defend against privilege redelegation [41, 52].

Enck et al. present TaintDroid, a modification to the Android platform to provide dynamic taint tracking on sensitive data (e.g., location, contact lists, etc.) [44]. Hornyack et al. present AppFence, a tool to provide users with the option to either prevent data from leaving the phone or provide false shadow data in place of legitimate data [57]. These systems focus on protecting user privacy by limiting the behavior of grayware and malware.

# Chapter 6 Conclusion

With new functionality comes new security threats. Android provides tools to enable rich interaction, but if developers do not know how to use them correctly, they will not use them securely. In this dissertation, we examined how mobile applications interact with each other and their environment and uncover threats to application security due to developer confusion and general misuse of the features provided by the mobile platform.

First, we examined how applications interact with other applications. We present two tools, ComDroid and IntraComDroid, to detect communication vulnerabilities. We found that over 60% of applications have communication-related vulnerabilities including but not limited to message sniffing, modification, theft, and replacement, and forgery. We further identified a subclass of vulnerabilities due to internal messages being unnecessarily broadcast publicly. We proposed a platform modification that would patch all of these vulnerabilities, which make up 31% of the original vulnerabilities identified. We also suggest a modification to the communication API to separate internal from external communication mechanisms.

Second, we examined how applications gain access to system resources. We present Stowaway a tool to identify applications that request more permissions than used by the application itself. We found that over one-third of applications are overprivileged. Developers mistake similar sounding permissions, make copy and paste errors, and include permissions that are required by related methods. This confusion can be addressed by simplifying the permission system and providing better API documentation.

Finally, we examined how applications interact with web content. We present Bifocals a tool to identify applications that are vulnerable to malicious web content. We found that over 11% of applications contain WebView-related vulnerabilities. We suggest a modification to WebView security policies that would protect over 60% of the vulnerable applications with little burden on developers.

By analyzing rich features and identifying ways developers go wrong, we can prevent security flaws and improve mobile systems. Through this work, we provide platform-level, API-level, and design-level solutions that can help developers and platform designers build secure applications and systems.

# Appendix A

## A.1 Manual Review

We manually reviewed all ComDroid warnings for the following applications:

Adobe Photoshop Express, App Protector Pro, Bubble, Halloween Live Wallpaper, ICE - In Case of Emergency, IMDb Movies & TV, Instant Heart Rate, Kindle for Android, Korean Nationwide Bus, Pageonce Pro - Money and Bills, PicSay Pro - Photo Editor, Retro Camera, Smart Keyboard PRO, Starlight Live Wallpaper, Steamy Window, SwiftKey Keyboard, Tango Video Calls, TweetCaster Pro for Twitter, Uninstaller, and WolframAlpha.

## A.2 Example Code

**Declaring Components** When declaring a component in the manifest, a developer can use the "exported" attribute to make the component explicitly internal (i.e., private):

```
<activity android:name=".TestActivity"
android:exported="false">
</activity>
```

If a developer needs to make a component selectively accessible to external applications, he or she can use permissions to restrict access to applications with the given permissions:

```
<activity android:name=".TestActivity2"
android:exported="true">
android:permission="my.permission">
<intent-filter>
<action android:name="my.action.TEST"/>
</intent-filter>
</activity>
```

If a component is protected with a new permission, the new permission can be declared as such:

```
<permission
android:description="My test permission"
android:name="my.permission"
android:protectionLevel="signature"/>
```

Alternately, the component implementation can dynamically call checkCallingPermission(String permission) to verify that the caller has the specified permission.

**Declaring Intents** When sending an Intent, the developer can make the recipient explicit by setting a destination class:

```
Intent i = new Intent();
i.setClassName("some.package.name", "some.package.name.TestActivity");
```

Or, equivalently, the class name can be set with one of the following three methods:

```
setClass(Context ctxt, Class<?> cls)
setClassName(Context ctxt, String className)
setComponent(ComponentName component)
```

Or it can be limited to be sent to components of a specific package:

```
setPackage(String packageName)
```

If a Receiver is intended to only accept system broadcast actions, then the developer should check the received action:

```
public void onReceive(Context ctxt, Intent i) {
    if (!i.getAction().equals("expected.action"))
        return;
    doSomething();
    ...
}
```

## A.3 System Broadcast Actions

Several examples of system broadcast actions:

```
android.intent.action.ACTION_POWER_CONNECTED
android.intent.action.ACTION_POWER_DISCONNECTED
android.intent.action.ACTION_SHUTDOWN
android.intent.action.BATTERY_CHANGED
```

android.intent.action.BATTERY\_LOW
android.intent.action.BATTERY\_OKAY
android.intent.action.BOOT\_COMPLETED
android.intent.action.CONFIGURATION\_CHANGED
android.intent.action.DEVICE\_STORAGE\_LOW
android.intent.action.DEVICE\_STORAGE\_OK

# Bibliography

- [1] Adblock plus. http://adblockplus.org/.
- [2] AdMarvel. http://www.admarvel.com/.
- [3] Android Market. http://www.android.com/market/.
- [4] Android permissions. http://android.git.kernel.org/?p=platform/frameworks/ base.git;a=blob;f=core/res/AndroidManifest.xml.
- [5] BlackBerry App World. http://na.blackberry.com/eng/services/appworld/.
- [6] CyanogenMod. http://www.cyanogenmod.com/.
- [7] Dashboards: Platform versions. http://web.archive.org/web/20130205234427/ http://developer.android.com/about/dashboards/index.html.
- [8] HTTP state management mechanism RFC. http://www.rfc-editor.org/rfc/ rfc6265.txt.
- [9] iPhone App Store. http://www.apple.com/iphone/apps-for-iphone/.
- [10] Malware delivered by Yahoo, Fox, Google ads. http://news.cnet.com/8301-27080\_3-20000898-245.html.
- [11] Malware-infected WinRAR distributed through Google AdWords. http: //www.zdnet.com/blog/security/malware-infected-winrar-distributedthrough-google-adwords/2405.
- [12] Medialets. http://www.medialets.com/.
- [13] Millennial media. http://www.millennialmedia.com/.
- [14] MobiStealth. http://www.mobistealth.com/.
- [15] nsIContentPolicy. https://developer.mozilla.org/en-US/docs/XPCOM\_ Interface\_Reference/nsIContentPolicy.

- [16] Old habits die hard: Cross-zone scripting in Dropbox & Google Drive mobile apps. http://blog.watchfire.com/wfblog/2012/10/old-habits-die-hard.html.
- [17] Security enhancements in Jelly Bean. http://android-developers.blogspot.com/ 2013/02/security-enhancements-in-jelly-bean.html.
- [18] Smali and baksmali. http://code.google.com/p/smali/.
- [19] Times web ads show security breach. http://www.nytimes.com/2009/09/15/ technology/internet/15adco.html.
- [20] Amazon Appstore for Android. http://www.amazon.com/mobile-apps/b/ref= topnav\_storetab\_mas?ie=UTF8&node=2350149011.
- [21] Android developers reference. http://developer.android.com/reference/.
- [22] Appventive. ICE: In case of emergency. http://www.appventive.com/ice.
- [23] S. Artzi, M. Ernst, A. Kiezun, C. Pacheco, and J. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In Proc. of 1st Workshop on Model-based Testing and Object-Oriented Systems (M-TOOS), 2006.
- [24] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie. Short paper: A look at smartphone permission models. In Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices (SPSM), 2011.
- [25] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In Proc. of the 19th ACM Conference on Computer and Communications Security (CCS), 2012.
- [26] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In Proc. of the 17th ACM conference on Computer and Communications Security (CCS), 2010.
- [27] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permissionbased software by reducing the attack surface: An application to Android. In Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012.
- [28] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In Proc. of the 15th ACM Conference on Computer and Communications Security (CCS), 2008.
- [29] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In Proc. of the 6th International Conference on Malicious and Unwanted Software (MALWARE), 2011.

- [30] E. Bodden, A. Sewe, J. Sinschek, and M. Mezini. Taming reflection: Static analysis in the presence of reflection and custom class loaders. Technical Report TUD-CS-2010-0066, CASED, Mar. 2010.
- [31] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In Proc. of the 6th International Conference on Mobile Systems, Applications, and Services, 2008.
- [32] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In Proc. of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, 2002.
- [33] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In Proc. of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), 2011.
- [34] J. Burns. Mobile application security on Android. *Blackhat*, 2009.
- [35] B. Chess and G. McGraw. Static analysis for security. Security & Privacy, IEEE, 2(6):76–79, 2004.
- [36] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet security: Repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.
- [37] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In Proc. of the Annual International Conference on Mobile Systems, Applications, and Services, 2011.
- [38] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. Software: Practice and Experience, 34(11):1025–1050, 2004.
- [39] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Information Security*, pages 346–360. Springer, 2011.
- [40] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In Proc. of the 6th IEEE International Workshop on Web Site Evolution (WSE), 2004.
- [41] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In Proc. of the 20th USENIX Security Symposium, 2011.
- [42] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In Proc. of the 20th ACM Symposium on Operating Systems Principles, 2005.

- [43] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proc. of the Network and Distributed System Security Symposium*, 2011.
- [44] W. Enck, P. Gilbert, B.-g. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010.
- [45] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In Proc. of the 20th USENIX Security Symposium, 2011.
- [46] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In Proc. of the 16th ACM Conference on Computer and Communications Security (CCS), 2009.
- [47] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. IEEE Security and Privacy, 7(1):50–57, 2009.
- [48] D. Endler. The evolution of cross site scripting attacks. Whitepaper, iDefense Inc., 2002.
- [49] Enhanced JUnit. http://www.silvermark.com/Product/java/enhancedjunit/ index.html.
- [50] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In Proc. of the 18th ACM conference on Computer and Communications Security, 2011.
- [51] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In Proc. of the 2nd USENIX Conference on Web Application Development, 2011.
- [52] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proc. of the 20th USENIX Security Symposium*, 2011.
- [53] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. Technical report, University of Maryland, 2009.
- [54] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.
- [55] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In Proc. of the 19th Annual Symposium on Network and Distributed System Security, 2012.
- [56] D. Hackborn. Re: List of private / hidden / system APIs? http://groups.google. com/group/android-developers/msg/a9248b18cba59f5a.

- [57] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In Proc. of the 18th ACM conference on Computer and Communications Security, 2011.
- [58] M. Howard, J. Pincus, and J. Wing. Measuring relative attack surfaces. Computer Security in the 21st Century, pages 109–137, 2005.
- [59] IMDb. IMDb Movies & TV. http://www.androlib.com/android.application.comimdb-mobile-jzEzw.aspx.
- [60] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In Proc. of the 2nd IEEE Communications Society/CreateNet International Conference on Security and Privacy in Communication Networks, 2006.
- [61] C. Kanich, N. Chachra, D. McCoy, C. Grier, D. Y. Wang, M. Motoyama, K. Levchenko, S. Savage, and G. M. Voelker. No plan survives contact: Experience with cybercrime measurement. In Proc. of the 4th Conference on Cyber Security Experimentation and Test, 2011.
- [62] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intraapplication communication in Android. In Proc. of the 2nd ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), 2012.
- [63] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. ScanDal: Static analyzer for detecting privacy leaks in Android applications. In *Proc. of the Mobile Security Technologies*, 2012.
- [64] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In Proc. of the 2006 ACM Symposium on Applied Computing, 2006.
- [65] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles, 2007.
- [66] H. Lee. Nationwide bus. http://www.androlib.com/android.application.nethyeongkyu-android-incheonbus-Eqwq.aspx.
- [67] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In Proc. of the 3rd Asian Symposium on Programming Languages and Systems, 2005.
- [68] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In Proc. of the 14th Conference on USENIX Security Symposium, 2005.
- [69] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS), 2012.

- [70] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on WebView in the Android system. In Proc. of the 27th Annual Computer Security Applications Conference, 2011.
- [71] P. Manadhata, J. Wing, M. Flynn, and M. McQueen. Measuring the attack surfaces of two FTP daemons. In Proc. of the 2nd ACM Workshop on Quality of Protection, 2006.
- [72] G. McCluskey. Using Java reflection. http://java.sun.com/developer/ technicalArticles/ALT/Reflection, 1998.
- [73] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology (TOSEM), 9(4):410–442, 2000.
- [74] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In Proc. of the ACM Conference on Computer and Communications Security, 2012.
- [75] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich applicationcentric security in Android. In Proc. of the 25th Annual Computer Security Applications Conference (ACSAC), 2009.
- [76] C. Pacheco and M. Ernst. Randoop. http://code.google.com/p/randoop/.
- [77] C. Pacheco and M. Ernst. *Eclat: Automatic generation and classification of test inputs.* Springer, 2005.
- [78] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In Proc. of the 29th international conference on Software Engineering, 2007.
- [79] G. Paller. Dedexer. http://dedexer.sourceforge.net/.
- [80] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In Proc. of 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2012.
- [81] J. Sawin and A. Rountev. Improving static resolution of dynamic class loading in Java using dynamically gathered environment information. Automated Software Engg., 16:357–381, 2009.
- [82] SC Magazine. WhiteHat: 90 percent of websites vulnerable to attack. http://www.scmagazine.com/whitehat-90-percent-of-websites-vulnerableto-attack/article/58066/.
- [83] R. Scandariato and J. Walden. Predicting vulnerable classes in an Android application. In Proc. of the 4th International Workshop on Security Measurements and Metrics, 2012.

- [84] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on Android. In *Proc. of International Conference on Communications (ICC)*, 2009.
- [85] Stack Overflow. Broadcast Intent when network state has changend. http:// stackoverflow.com/q/2676044.
- [86] Stack Overflow. Developer sites contradict each other regading webviewshouldoverrideurlloading. http://stackoverflow.com/q/10865788.
- [87] M. A. Troy Vennon. Android malware: Spyware in the Android Market. Technical report, SMobile Systems, March 2010.
- [88] T. Vennon. Android malware: A study of known and potential malware threats. Technical report, SMobile Systems, February 2010.
- [89] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In Proc. of the Web, 2011.
- [90] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Proc. of Network and Distributed System Security Symposium, 2000.
- [91] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In Proc. of the 7th Symposium on Operating Systems Design and Implementation, 2006.
- [92] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In Proc. of the 19th Annual Network and Distributed System Security Symposium, 2012.