Automatic Detection of Inter-application Permission Leaks in Android Applications

Dragoș Sbirlea∗ 1, Michael G. Burke † 1, Salvatore Guarnieri ‡ 2, Marco Pistoia§ 2, and Vivek Sarkar¶ 1

1Department of Computer Science, Rice University
2IBM Watson Research Center

Abstract

The Android operating system builds upon already well established permission systems, but complements them by allowing application components to be reused within and across applications through a single communication mechanism, called the Intent mechanism. In this paper we develop techniques for statically detecting Android application vulnerability to attacks that obtain unauthorized access to permission-protected information. We address three kinds of such attacks, known as confused deputy, permission collusion, and Intent spoofing.

We show that application vulnerability to these attacks can be detected using taint analysis. Based on this technique, we developed PermissionFlow, a tool for discovering vulnerabilities in the bytecode and configuration of Android applications. To enable the PermissionFlow analysis, we develop a static technique for automatic identification of permission-protected information sources in permission-based systems. This technique identifies APIs whose execution leads to permission-checking and considers these APIs to be sources of taint. Based on this approach, we developed Permission Mapper, a component of PermissionFlow that improves on previous work by performing fully automatic identification of such APIs for Android Java code.

Our automated analysis of popular applications found that 56% of the top 313 Android applications actively use inter-component information flows. Among the tested applications, PermissionFlow found four exploitable vulnerabilities. By helping ensure the absence of inter-application permission leaks, we believe that the proposed analysis will be highly beneficial to the Android ecosystem and other mobile platforms that may use similar analyses in the future.

1 Introduction

Users of modern smartphones can install third-party applications from markets that host hundreds of thousands of applications [24, 1] and even more from outside of official markets. To protect sensitive user information from these potentially malicious applications, most operating systems use permission-based access-control models (Android [12], Windows Phone 7 [18], Meego [26] and Symbian [19]).

Permissions are a well known and powerful security mechanism, but - as with any new operating system - there is the possibility that Android-specific features may reduce the guarantees of the classic permissions model. One such feature is the new communication mechanism (called Intents), which can be used to exchange information between components (called Activities) of the same application or of different applications.

One type of attack that exploits Intents for malicious purposes is permission collusion. In this attack, an application that individually only has access to harmless permissions augments its capabilities by invoking a collaborating application through sending and receiving Intents. To stage this attack, malevolent developers could trick users into installing such cooperating malicious applications that covertly compromise privacy.
A second type of attack using **Intents** is the *confused deputy attack*. Confused deputy attacks rely on misconfigured applications; components that interact with other applications are invoked by unauthorized callers and allow them to perform protected actions or access permission-protected information.

A third type of attack, **Intent spoofing** [4], is an Android-specific form of the confused deputy attack: it affects applications not meant to communicate with other applications. Even if a developer’s intention was to disallow external invocation of internal Activities, other applications may be able to invoke them if the application does not have the necessary configuration. This is possible because **Intents** can be used for inter-application invocations as well as intra-application invocations.

In this paper we focus on the use of the above types of attacks to obtain unauthorized access to permission-protected information via exploiting the **Intent** mechanism. We call these attacks **permission-leak attacks**.

Tested on 313 popular Android Market applications, our tool, PermissionFlow, identified that 56% of them use inter-component information flows that may require permissions. Four exploitable vulnerabilities were found. Our contributions are the following:

- We describe a static analysis-based technique that detects permission-leaking **Intent** vulnerabilities in Android applications. Based on this technique, we developed PermissionFlow, a tool for discovering vulnerabilities in the bytecode and configuration of Android applications.

- We propose a static analysis-based technique for automatic identification of permission-protected information sources in permission-based systems. Our approach consists of identifying APIs whose execution leads to permission-checking and considering these APIs to be sources of taint. Based on this approach, we developed Permission Mapper, a component of PermissionFlow that improves on previous work by performing fully automatic identification of such APIs for Android Java code.

- We evaluate PermissionFlow on leading Android applications and show that a majority (177 out of 313) applications tested use **Intents** to invoke **Activities** that return information. These applications could benefit from PermissionFlow to ensure that the use of this feature is secure. PermissionFlow found three permission-protected leaks in widely used applications and an additional vulnerability that allows leaking of information that should be protected by custom permissions.

2 Background

The vulnerabilities we identify involve knowledge about the Android development model, the Android inter-process communication mechanism and its permissions system. These components are the focus of the following subsections.

2.1 Android development

Android applications are typically written in Java using both standard Java libraries and Android-specific libraries. On Android devices, the Java code does not run on a standard JVM, but is compiled to a different register-based set of bytecode instructions and executed on a custom virtual machine (Dalvik VM). Android application packages, also called **APKs** after their file extension, are actually ZIP archives containing the Dalvik bytecode compiled classes, their associated resources such as images and the application manifest file.

The **application manifest** is an XML configuration file (AndroidManifest.xml) used to declare the various components of an application, their encapsulation (public or private) and the permissions required by each of them.

Android **APIs** offer programmatic access to mobile device-specific features such as the GPS, vibrator, address book, data connection, calling, SMS, camera, etc. These APIs are usually protected by permissions.

Let’s take for example the **Vibrator** class: to use the `android.os.Vibrator.vibrate(long milliseconds)` function, which starts the phone vibrator for a number of milliseconds, the permission `android.permission.VIBRATE` must be declared in the application manifest, as seen on line 2 of Listing 1.

**Application signing** is a prerequisite for inclusion in the official Android Market. Most developers use self-signed certificates that they can generate themselves, which do not imply any validation of the identity of the developer. Instead, they enable seamless updates to applications and enable data reuse among sibling
applications created by the same developer. Sibling applications are defined by adding a `sharedUid` attribute in the application manifest of both, as seen in line 1 of Listing 1.

**Activities.** The Android libraries include a set of GUI components specifically built for the interfaces of mobile devices, which have small screens and low power consumption. One type of such component is Activities, which are windows on which all visual elements reside. An Activity can be a list of contacts from which the user can select one, or the camera preview screen from which he or she can take a picture, the browser window, etc.

**Intents.** To spawn a new Activity, programmers use Intents and specify the name of the target class, as shown in the following snippet:

```java
Intent i = new Intent();
i.setClassName( this, "package.CalleeActivity" );
startActivity(i);
```

Usually the parent Activity needs to receive data from the child Activity. This is possible through the use of Intents with return values. The parent spawns a child by using `startActivityForResult()` instead of `startActivity()` and is notified when the child returns through a callback (the `onActivityResult()` function), as shown in Listing 2. This allows the parent to read the return code and any additional data returned by the child Activity.

```java
void onActivityResult( int requestCode, int resultCode, Intent data) {
    if (requestCode == CREATE_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            String info = intent.getStringExtra("key");
        }
    }
}
```

Listing 2: Code snippet showing how a caller accesses information returned by a child Activity.

```java
Intent intent = new Intent();
intent.putExtra("key", "my value");
this.setResult(RESULT_OK, intent);
finish();
```

Listing 3: Code snippet showing how child Activities can return data to their caller.

As shown in Figure 3, the child Activity needs to call the `setResult` function, specifying its return status. If additional data should be returned to the parent, the child can attach an Intent along with the result code and supply extra key/value pairs, where the keys are Java Strings and the values are instances of Parcelable types, which are similar to Java Serializable classes and include Strings, arrays and value types.
Sending **Intents** to explicitly named **Activities**, as described above, is called explicit **Intent** usage. Android also allows creation of **Intents** specifying a triple (action, data type, category) and any **Activity** registered to receive those attributes through an **intent-filter** will be able to receive of the **Intent**. If there are multiple **Activities** that can receive the **Intent**, the user will be asked to select one.

The explicit **Intent** feature is mostly used in intra-application communication, as described in the following section, but can be useful for inter-application communication too and its existence is the root cause of the vulnerabilities discovered by us.

### 2.2 Inter-application Intents and data security

**Inter-process communication with Intents.** **Intents** can be used for communication between **Activities** of the same application or for inter-application communication. In the second case, **Intents** are actually inter-process message-passing primitives. To specify a subset of **Intents** that an **Activity** answers to, developers add to the application manifest an **intent-filter** associated with the **Activity**. The **intent-filter** in Figure [1] specifies that **MyActivity** can be invoked by sending an **Intent** with action `com.zxing.SCAN`; such an **Intent** is called an implicit **Intent** because it does not specify a particular **Activity** to be invoked. Implicit **Intents** are created using the single parameter constructor `new Intent(String)`.

**Component encapsulation.** Developers enable or disable inter-application invocation of their **Activities** by setting the value of the boolean **exported** attribute of each **Activity** in the application manifest. The behavior of this attribute is a detail that may be a source of confusion, as the meaning depends on the presence of another XML element, the **intent-filter**:

- If an **intent-filter** is declared and the **exported** attribute is not explicitly set to **true** or **false**, its default value is **true**, which makes the **Activity** accessible by any application.
- If an **intent-filter** is not declared and the **exported** attribute is not set, by default the **Activity** is only accessible through **Intents** whose source is the same application.

An exception to the above rules is allowed if the developer specifies the attribute **sharedUid** in the manifest file. In that case, another application may run in the same process and with the same Linux user ID as the current application. This addition changes the behavior of **Activities** that are not exported: they can be invoked not only from the same application, but also from the sibling application with the same user ID. Listing [1] shows the use of the **sharedUserId** attribute.

It is important to realize that the **intent-filter** mechanism does not provide any security guarantees and is meant only as a loose binding between **Activities** and **Intents**; any **Activity** with an **intent-filter** can still be sent an explicit **Intent** in which case the **intent-filter** is ignored. The presence of this attribute, however, changes the behavior of the security-related **exported** attribute, as detailed above. We found that many developers overlook the security-related implications when using **intent-filters**.

### 3 Attacks on permission-protected information

All Android applications with a graphical user interface contain at least one **Activity**, which means vulnerabilities related to **Activities** can affect a majority of applications. All the vulnerabilities that we identify have in common the existence of information flows that are meant to allow child **Activities** to communicate with authorized parents, but can instead be used by unauthorized applications to access sensitive information without explicitly declaring the corresponding permission.

We consider three different attack scenarios, discussed in the following paragraphs and our tool, PermissionFlow, identifies the flow vulnerabilities that enable all of them. PermissionFlow validation can be used as a prerequisite for applications before being listed in Android Market and by developers to ensure the security of their applications or by users.

Note that other operating systems sandbox applications and do not offer a mechanism for direct inter-application communication; in spite of this, some of these attacks are still possible. For example, in iOS, the colluding applications attack can be performed through URL Schemes [5]. For these attacks to be possible, certain misconfigurations have to exist. These misconfigurations consist of a combination of implicitly public **Activities**
Figure 1: Possible attacks: internal Activity invocation or confused deputy, application sharing user ID with a compromised application (center) and permission collusion by malevolent applications (right).

(callable by unexpected callers) and misconfiguration of Activity permissions, which consists of failure to enforce the ownership of permissions on callers for Activities that return permission-protected information. [*]

1. **Attacks on misconfigured applications** happen when an attacker application installed on the device can exploit the flows of a misconfigured application. If an application has any one of the configuration parameter combinations listed in Table 1 as high risk, then any application on the device can spawn it.

If in the application manifest an Activity is listed with an intent-filter and is not accompanied by a exported= "false" attribute, any other application on the system can invoke it. Then, in the absence of declarative or dynamic permission checking by the developer, information returned to the caller through the Intent result may compromise permission-protected information, as no permission is required of the caller.

In the example from Figure 1 (left), the user installed malevolent application B (a music streaming app) with permission to access the Internet. B can exploit the honest but misconfigured contact manager application A by invoking Activity A2 that returns the contacts; B can then send the contacts to a remote server. If A2 is built to reply to external requests and it just failed to check that B has the proper permission, then the attack is a classic confused deputy attack. However, because in Android Intents are also used as an internal (intra-application) communication mechanism, it is possible that A2 is not built for communicating with another application and is just misconfigured. This Intent spoofing is a more powerful attack than confused deputy for two reasons. First, it targets internal APIs, not just public entry points. These internal APIs are generally not regarded as vulnerable to confused deputy and so not they are not secured against it. By increasing the number of APIs that can be targeted, this attack increases the likelihood that the returned information is permission protected. (Protected information tends to flow between internal components such as A1 and A2, even when it does not leave the application.) Second, the problem in this attack is not that the deputy performs a protected operation, but that it sends protected information to the callee.

For Activities that are designed to be invoked by unknown applications, developers can ensure that callers own a set of permissions in one of two ways: declaratively (in the manifest file, using the permission attribute of the Activity) or dynamically (by calling the function checkCallingPermission( String permission)). Note that the permission attribute can only be used to enforce a single permission and is different from the uses-permission node in Listing 1 which controls what permissions the application needs in order to function.

The safest approach is to completely disable outside access to internal Activities that may leak protected information. Table 1 shows the combinations of configuration parameters that may lead to information leaks.

2. **Collusion attacks** obtain permission-protected information without requesting the permission, by exploiting the combination of assignment of Android permissions on a per-application basis and the exchange of applications information without making this explicit to the user.

In Figure 1 (right), we show a scenario in which a user is tricked by a malevolent developer MD into installing two separate applications, that seem to have little risk associated with them. For example, a camera application that does not require the Internet permission seems safe, as it cannot upload the pictures to the Internet. Similarly, a music streaming application that does not request the Camera permission would be acceptable. However, if the two applications are malicious, the music streaming application can invoke the camera application and send the pictures obtained from it remotely. The Android security system does not inform the user of this.

---

1 Previous work considered implicitly public Activities, but did not but did not test if those Activities check for permissions of their callers, leading to some trustworthy applications being considered vulnerable (false positives).
<table>
<thead>
<tr>
<th>Activity configuration</th>
<th>Application configuration</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported</td>
<td>Intent-filter</td>
<td>SharedUid</td>
</tr>
<tr>
<td>exported=&quot;true&quot;</td>
<td>present</td>
<td>any</td>
</tr>
<tr>
<td>exported=&quot;true&quot;</td>
<td>absent</td>
<td>any</td>
</tr>
<tr>
<td>exported=&quot;false&quot;</td>
<td>present</td>
<td>set</td>
</tr>
<tr>
<td>exported=&quot;false&quot;</td>
<td>absent</td>
<td>set</td>
</tr>
<tr>
<td>default</td>
<td>present</td>
<td>any</td>
</tr>
<tr>
<td>default</td>
<td>absent</td>
<td>set</td>
</tr>
</tbody>
</table>

Table 1: Different configurations lead to different levels of vulnerability.

application collusion risk.

3. Attack on applications sharing the user ID. We have not yet discussed an additional type of attack that our approach can recognize, but is improbable in practice because of its narrow applicability. This type of attack is on sibling applications.

The vulnerability in this case allows an attacker to access the permission-protected information of applications sharing the user ID with the already compromised N. If N is configured to have the same user ID as application M (as shown in Figure 1 center), it can then obtain the information from M. To set up this attack, an attacker would need to control application N of developer D. N should have any exfiltration permission (sending short messages, accessing the Internet, etc.) and N should share user ID with an application by D that returns permission-protected information. Then, N can invoke most Activities of M, even if M is configured according to the rows with low risk level in Table 1. PermissionFlow can detect this vulnerability too.

4 System description

PermissionFlow has two main parts. The first one is a general, reusable taint analysis framework; the second consists of all other components, which are Android-specific.

To analyze real Android Market applications, whose source code is usually not available, we support input in the form of Android binary application packages (APK files). This means PermissionFlow can also be used by Android users, developers and security professionals.

The system design (Figure 2) consists of the following components:

- The Permission Mapper (labelled 1 in the Figure 2) builds a list of method calls in the Android API that require the caller to own permissions. Its inputs are Android classes obtained by building the Android source code, with any modifications or additions performed by the device manufacturer. Having the complete system code as input allows the mapper to extract all the permissions-protected APIs that will be present on the device. It builds a permissions map, which maps permission-protected methods to their required permissions.

- The permissions map is passed to the Rule Generator, which builds the taint analysis rules relating the sources in the map with their corresponding sinks. In our case, the only sink is the Activity.setResult method with an Intent parameter.

- Our taint analysis engine, (labelled 3) reads the generated rules and any extra rules manually added for detecting application-dependent private information. It outputs the flows that take the protected information from sources to sinks. For this it needs access to the application classes and the Android library classes. The taint analysis engine also needs access to the Android library, in order to track flows that go through it, for example callbacks that get registered, Intents that get passed to the system, etc.

- The dex2jar decompiler [23] (labelled 4) is used to extract from the application APK a JAR archive containing the application bytecode.

- To extract the binary application manifest from the application package we use the ApkTool [21] (labelled 5); the decompilation step needed to get the textual XML representation is performed by AXMLPrinter2 [22] (labelled 6).
• The taint analysis engine outputs the flows from sources to sinks if there are any, but the presence of flows does not in itself imply that the application is vulnerable. The **Decision Maker** (labelled 7) looks for the patterns identified in Table 1 in the application manifest file - these patterns correspond to misconfigurations that allows successful attacks to take place. If an application contains a vulnerable information flow and is improperly configured, only then is it vulnerable. It is improperly configured if it is public (as shown in Table 1) and fails to enforce on its callers the permissions protecting the information it returns.

4.1 The Permission Mapper

The Permission Mapper matches function calls used for permission enforcement in the Android libraries to Android library functions that use these calls. In short, it uses static analysis to identify permission-protected methods and to map them to their required permissions; this analysis is independent of any application analysis and needs to be performed only once for each input Android configuration.

Identifying sources of permission-protected information could also be attempted by crawling the documentation. However, it is incomplete even for public, documented classes and does not include public, but non-documented methods and does not account for Java reflection on non-public methods. It also does not account for any modifications and additions to the Android API performed by the phone manufacturer. Felt et al. [9] showed that it is possible to identify which API calls require permissions though a combination of automated testing and manual analysis, but they use techniques that allow false negatives, need partial manual analysis and do not handle the version and feature fragmentation [11] of Android well.

For these reasons, we built the Permissions Mapper, a reliable and automatic tool for identifying permission-protected APIs and their required permissions.

Most Android permissions are enforced through calls to the `checkPermission` function of the `Context` and `PackageManager` classes or the `checkCallingOrSelfPermission` function of the `Context` class; our work targets complete coverage of APIs enforced through this mechanism, which includes the majority of Android permissions. Other APIs are enforced through native code or Linux users which we do not consider.

To illustrate how permission checks work, we can use for example the `VIBRATE` permission. To use the phone vibrator, an application needs to own the `VIBRATE` permission; all functions that require this permission check for it.

When this function is called, the Android API forwards the call to a system service. The service process is the one that makes the actual permission checks, before performing any protected operation, as shown in...
```java
public class VibratorService extends IVibratorService.Stub {
    public void vibrate(long milliseconds, binder) {
        if (context.checkSelfPermission(VIBRATE) != PackageManager.PERMISSION_GRANTED) {
            throw new SecurityException("Requires VIBRATE permission");
        }
    }
}
```

Listing 4: Code snippet showing how services check the permissions of the application.

Figure 3: The permission analysis exemplified on the vibrate() call. The call graph edge from the proxy to its corresponding service is automatically added before the analysis.

Listing 4. The proxy for the service performs the inter-process communication and because of this it appears as a leaf in the call graph. In our analysis, we automatically fill in the missing edges between proxies and their corresponding services (relying on the name correspondence between the two, which is enforced by the Android AIDL code generation).

The interprocedural dataflow analysis is built using IBM WALA and starts by building the call graph of the Android libraries, including all methods as entry points. All call chains containing a Context.checkPermission(String) method call are then identified. To find the actual permission string that is used at a checkPermission call site, we follow the def-use chain of the string parameter. Once found, we label all callers upstream in the call chain as requiring that permission. Note that different call sites of checkPermission will have different permission strings - each such string needs to be propagated correctly upstream, building set of required permissions for each function.

For the vibrate() example, the permission enforcement call chain is in Figure 3. The string value of the vibrate permission is located by following the def-use chain of the checkPermission parameter (dashed lines) until the source String constant is found. Once the constant is found, we need to identify which functions on the call chain need this permission. We start by labeling the function that contains the first (“most downstream”) call site through which the def-use chain flows. In our case, the def-use chain goes first through the call site of checkCallingOrSelfPermission in IVibratorService.vibrate (line 4 in Figure 4), where the VIBRATE variable is specified as a parameter, so IVibratorService is labeled with “android.permission.VIBRATE”.

After labeling IVibratorService.vibrate, the same label must be propagated to callers of that function, but in our case there are no callers except the proxy stub.

Because the communication between Android proxy stubs and their corresponding services (shown as dotted edges in Figure 3) is done through message passing, it does not appear in the actual call chain built by WALA. To work around this problem we add the permissions labels of the service methods to the corresponding proxies; the labels are then propagated to any callers of those methods.
4.2 Taint Analysis Engine

For the taint analysis engine, we used Andromeda [27], which is highly scalable and precise as well as sound; it builds on IBM WALA [28]. Andromeda uses as input rules composed of two sets: sources and sinks. The sources are parameters and return values of functions that are the origin of tainted data and the sinks are security-critical methods. The engine tracks data flow from the source through assignments, method calls, and other instructions, until the data reaches a sink method. If the taint analysis engine discovers that tainted data reaches a sink method, the flow is included in the taint analysis engine output. We discuss the soundness of the complete PermissionFlow analysis in Section 5.2.

5 Experimental results

5.1 Evaluation of the permissions map

We evaluated the Android API Permissions Mapper by directly comparing its output permissions map with that of Felt et al. [9], based on automated and manual testing. To perform the comparison with the work by Felt et al., we eliminated permissions from their map that are enforced through mechanisms other than the checkPermissions calls, because our analysis only targets checkPermissions-enforced permissions.

Comparing the size of their reference map (which includes 1311 calls that require permissions) with ours (4361 calls with permissions) shows that our tool finds more functions that require permissions. The larger size of our map is partly explained by the lack of false negatives for the analyzed Java APIs. However, a direct comparison is not possible as the input classes on which Felt et al. ran their analysis is not specified in their paper.

Through manual comparison, we identified one false negative in their map (probably due to the automated testing not generating a test and the subsequent analysis not detecting the omission). The function is MountService.shutdown, which usually needs the SHUTDOWN permission, but if the media is shared it also needs the MOUNTUnmount_FILESYSTEMS permission. The existence of missing permissions in the testing-based methods shows that testing methods, even if party automated and enhanced by manual analysis, cannot offer guarantees with respect to false negatives.

Another reason for the higher number of methods found in our map is the existence of false positives: permissions that are reported as required but are not. We have identified the following sources of false positives, all of which are known weaknesses of static taint analysis:

- Checking for redundant permissions. For example checking for ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION in TelephonyManager.getCellLocation(), where either one is sufficient for enforcement; our method reports both, because it is oblivious to control flow.

- Data dependent checks. For example, a check for the VIBRATE permissions depends on the value of a parameter such as in NotificationManager.notify().

- Android provides the pair of functions clearIdentity and restoreIdentity that are used to change all checks so that they are performed on the service instead of the application using the service.

Another advantage of testing-based analysis is that it can cover areas, such as the permissions enforced in native code, which static analysis does not target (such as RECORD_AUDIO).

To perform a more detailed comparison of our approach with the work by Felt et al., we compared results obtained for a simple security analysis, identification of overprivileged applications, based on the permission map. This analysis consists of identifying Android applications that request in their manifest more permissions than they actually need to perform their functions. The results are used to reduce the attack surface of applications by removing unused permissions from the manifest. To perform this analysis, we built another static analysis tool based on IBM WALA, which records the API calls that can be performed by an application and computes, based on the permissions map, the permissions required by that application. The set of discovered permissions is then compared to the set of declared permissions.

We used both permission maps as input for the analysis of the Top Android Market Free Applications (crawled in December 2011) that were compatible with Android 2.3 and available in the US (354 applications). For a fair comparison, we removed from Felt et al.’s reference map the parts that relates content provider databases to their permissions, as these were outside the scope of our work. After eliminating applications that crashed the
We tested PermissionFlow on the same applications used to evaluate the permission map. To confirm the Intent.getExtra we taint the information returned by the Activity.setResult with an Activity. For inter-application flows and accepting some false positives but missing flows based on native code or Linux permissions, it enforces the correct permissions on the current (caller) application.

These 56% of the applications may be vulnerable if they also contain flows from taint sources to sinks and are consistent with the difference in the map sizes.

Out of the set of 313 applications, we found that three are vulnerable to permission-protected information leaks: Adobe Photoshop Express, SoundTracking and Sygic GPS. Adobe Photoshop Express and Soundtracking allow any application to send Intents to components that return permission-protected information, so their risk is high. Sygic can only leak such information to applications with the same user ID, so the risk level is low. The permission information compromised is the user contacts (Adobe Photoshop Express), the location from a different application, that application is assumed to be validated with PermissionFlow, so that it enforces the correct permissions on the current (caller) application.

To check for correctness, we ran PermissionFlow with our permissions map and the one produced by Felt. Using the map from Felt, PermissionFlow correctly identified two applications as vulnerable and had no false positives. With the permissions map built by our analysis, PermissionFlow outputs a larger set of vulnerable applications, but the additional applications are all false positives. As we saw in the previous section both permissions maps are incomplete: ours does not track permissions enforced through non-Java mechanisms and Felt’s allows the possibility of missing permission checks. Choosing one of the two maps amounts to either using a possibly incomplete map (Felt, et al.) and finding no false positives, or identifying the complete set of Java-based flows and accepting some false positives but missing flows based on native code or Linux permissions checks.

Our analysis is sound with respect to the subset of Android which we consider, which includes Activities. To maintain soundness with respect to chains of communicating Activities (from the same or different applications), we taint the information returned by the Intent.getExtra family of calls with the set of permissions owned by the current application. We need to guarantee that the incorporation of other Android components does not break the soundness of our analysis. To ensure security, these components should individually maintain the following invariant: if any information protected by permission P flows into the component, then the component must enforce permission P, through its manifest, on any components that read that information. This invariant can be checked in the same manner that PermissionFlow checks Activities (on each component individually).

To support Services more sinks need to be considered other than Activity.setResult(). For inter-application ContentProviders, we need access to the manifest that declares them, but we still do not need to analyze the source code of both simultaneously. None of these changes alters the dataflow analysis itself, only its input and output. Thus we maintain soundness when using a sound taint analysis engine such as Andromeda.

However, 47 permissions were identified as used by us and as unused by Felt et al. - a false positive rate of 4.8%.

PScout is a permission mapper built through static analysis based on Soot. The number of overpriviledged applications found by PScout is consistent with both Felt et al. and our tool. If we compare the number of entries in the map, PScout finds 17,218 APIs, we find 4,361 and Felt et al. list 1311. This inconsistency comes from four sources. First, different versions of Android are analyzed by each. Second, different subsets of classes may be selected as input from the Android code. Third, the tools have different levels of false positives (Felt has none, because it is a dynamic analysis, but PScout and our mapper may have some). Fourth, there are different levels of false negatives (Felt may have missing permissions listed because of incomplete coverage).

PScout reports a false positive rate of 7% on real applications whereas we have a rate of 4.8%. Just like our analysis, this tool omits permissions enforced through non-Java mechanisms. It includes permissions required for ContentProvider access, which we do not consider in our mapper (Section 5.2 explains how to include ContentProviders and Services in the PermissionFlow analysis).

The technical report contains a comparison of the Permission Mapper to a similar tool by Bartel et al.

5.2 Evaluation of PermissionFlow

We tested PermissionFlow on the same applications used to evaluate the permission map. To confirm the correctness of the results we manually inspected all applications. Out of the 313 applications, 177 use the Activity.setResult with an Intent parameter to communicate between components (both internal or external). These 56% of the applications may be vulnerable if they also contain flows from taint sources to sinks and are not configured properly. They can use PermissionFlow to check that they are secure.

The false positive rate is not a clear indicator, as the input applications used for the two systems were not the same, but it is consistent with the difference in the map sizes.

Our analysis is sound with respect to the subset of Android which we consider, which includes Activities. To maintain soundness with respect to chains of communicating Activities (from the same or different applications), we taint the information returned by the Intent.getExtra family of calls with the set of permissions owned by the current application. We need to guarantee that the incorporation of other Android components does not break the soundness of our analysis. To ensure security, these components should individually maintain the following invariant: if any information protected by permission P flows into the component, then the component must enforce permission P, through its manifest, on any components that read that information. This invariant can be checked in the same manner that PermissionFlow checks Activities (on each component individually). To support Services more sinks need to be considered other than Activity.setResult(). For inter-application ContentProviders, we need access to the manifest that declares them, but we still do not need to analyze the source code of both simultaneously. None of these changes alters the dataflow analysis itself, only its input and output. Thus we maintain soundness when using a sound taint analysis engine such as Andromeda.

Out of the set of 313 applications, we found that three are vulnerable to permission-protected information leaks: Adobe Photoshop Express, SoundTracking and Sygic GPS. Adobe Photoshop Express and Soundtracking allow any application to send Intents to components that return permission-protected information, so their risk is high. Sygic can only leak such information to applications with the same user ID, so the risk level is low. The permission information compromised is the user contacts (Adobe Photoshop Express), the location

The false positive rate is not a clear indicator, as the input applications used for the two systems were not the same, but it is consistent with the difference in the map sizes.

PermissionFlow already batches applications. This would be sufficient for a safe ContentProvider analysis.
(SoundTracking) and the camera pictures (Sygic). All of the vulnerabilities lead to a window being displayed to the user, corresponding to the invoked Activity from the callee application. If the user performs some activity in the window (selecting a contact, taking a picture), that information will leak. The window that pops up is familiar and belongs to a trusted application, so the user may unwittingly cooperate. Further, even though we did not find any, there may be other applications for which attacks do not require user intervention. See the technical report [25] for more details.

As examples of confidentiality violations that PermissionFlow identifies when provided with application-specific sources of taints, the GO SMS and GO Locker applications leak the lockscreen password. Since they have an identical pattern, we consider these GO applications to be the fourth vulnerability identified by the PermissionFlow tool. More information about the specific information flows that lead to these vulnerabilities can be found in the technical report [25].

6 Related Work

Privilege escalation attacks on Android applications have been previously mentioned in literature [6]. However, such an attack requires usage of native code, careful identification of buffer overflow vulnerabilities and high expertise. We focus only on vulnerabilities specific to Android and help protect the information before such attacks happen.

Michael Grace et al. [13] focused on static analysis of stock Android firmware and identified confused deputy attacks that enable the use of permission-protected capabilities. Our analysis is complementary in that it identifies not actions that are performed, but information that flows to attackers. Also we focus not on stock applications, but third-party applications.

TaintDroid [7] uses dynamic taint tracking to identify information flows that reach network communication sinks. Both PermissionFlow and TaintDroid can potentially support other sinks, and their dynamic approach is complementary to our static approach because it can better handle control flow (for example, paths that are never taken in practice are reported as possible flows by our tool). It can also enforce only safe use of vulnerable applications by denying users the capability to externalize their sensitive information.

SCanDroid [10] is the first static analysis tool for Android and can detect information flow violations. The tool needs to have access to both the vulnerable application and the exploitable application. To the best of our knowledge, SCanDroid is not easily extensible with new taint propagation rules.

CHEX [16], a system developed concurrently with our work, relies on static analysis to discover permission leaks in Android applications. CHEX uses an IR similar to the one used by WALA, but does not use WALA dataflow analysis. For efficiency, they use a graph reachability analysis. Andromeda, our taint analysis engine, achieves efficiency through use of a demand-driven taint analysis. CHEX detects several types of vulnerabilities affecting Android applications, including permission-protected information leaks. However, CHEX does not check the application manifest to identify if Activities are exported or if Activities use the manifest to enforce permissions from their callers. CHEX requires a permissions map as input, in that it does not automatically generate it.

ComDroid [4] is a tool that analyses inter-application communication in Android. ComDroid does not track permission-leak vulnerabilities. None of the vulnerabilities described pertain to permission-protected information. ComDroid does emit misconfiguration warnings, but these are not necessarily vulnerabilities (some applications offer public services that need no checking). Permission-leak attacks are a special case of Intent spoofs in that they imply permission-protected information flow, not only control flow and misconfiguration. Contributions such as automatic rule generation and automatic permission map building separate our work from theirs.

Kirin [20,8] is a tool based on a formal representation of the Android security model that checks if applications meet security policies. It can check for confused deputy vulnerabilities ("unchecked interface"), Intent spoofing ("intent origin") and other attacks by using a powerful Prolog-based security policy enforcement mechanism, which takes into consideration the set of applications already installed on a device. The authors point out several difficulties with creating information flow policies in Android and discuss the future possibility of including source code analysis to make information flow policies for Android practical. If we consider the PermissionFlow rules as information flow security policies, then PermissionFlow is a step in the right direction for such a tool. It would solve the problem they mention of information flowing into any application with a user interface.

Felt et al. [9] perform a similar analysis to our PermissionMapper. Their work is based on automated testing rather than static analysis, which means incomplete coverage and the possibility of false negatives in the permissions map. They do not use the map to check for information flow-based vulnerabilities in applications.
Their work is discussed further in Section 4.1. PScout [2] builds a permission mapper through static analysis based on Soot. Their approach is discussed in detail in Section 5.1.

Kantola et al. [15] propose to modify Android configuration semantics to implicitly mark fewer Activities as public. Their heuristic-based approach fixes most vulnerabilities and maintains backwards compatibility with applications written for the current Android semantics. The cost is the continuing reliance on heuristics, and vulnerabilities are still possible. A cleaner approach would be to disallow inter-application invocations if the caller does not own the permissions required by the callee. Such an approach prevents these vulnerabilities by correctly configuring applications, but breaks backwards compatibility. Another solution is to emit warnings based on the configuration and static analysis, which PermissionFlow would be well-suited for. In the technical report we provide recommended practices for safe application configurations for security-aware Android developers.

Hornyack et al. [14] describe a tool that can be used to complement ours. Mann and Starostin [17] propose a wider analysis based on typing rules that can discover flow vulnerabilities. We discuss both papers in the technical report.

7 Conclusion

This paper proposes an automated solution for the problem of checking for leaks of permission-protected information; this is an important security problem for mobile devices, as such leaks compromise users’ privacy. We demonstrate the benefits of this analysis on Android, and identify the Intent mechanism as a source of permission leaks in this operating system; we found that permissions can leak to other applications even from components that are meant to be private: i.e., accessed only from inside the application.

Our automated analysis of popular applications found that 56% of the top 313 Android applications actively use inter-component information flows. Among the tested applications, PermissionFlow found four exploitable vulnerabilities. Because of the large scale usage of these flows, PermissionFlow is a valuable tool for security-aware developers, for security professionals and for privacy-conscious users. Our approach extends beyond Android, to permission-based systems that allow any type of inter-application communication or remote communication (such as Internet access). Most mobile OSes are included in this category and can benefit from the proposed new application of taint analysis. By helping ensure the absence of inter-application permission leaks, we believe that the proposed analysis will be highly beneficial to the Android ecosystem and other mobile platforms that may use similar analyses in the future.

References


Biographical Sketches

Dragoş Sb˘ irlea Department of Computer Science, Rice University 6100 Main Street, Houston, Texas, 77005 (dragos@rice.edu). Mr. Sb˘ irlea is a Ph.D. Candidate in the Habanero Multicore Software Research group at Rice University. He is the author or co-author of 5 scientific publications. His research interests include analysis and optimization of parallel programs. He is a contributor to the Sandia Qthreads library and a member of ACM (Association for Computing Machinery).

Michael G. Burke Department of Computer Science, Rice University 6100 Main Street, Houston, Texas, 77005 (mgb2@rice.edu). Dr. Burke has been a Senior Research Scientist in the Computer Science Department at Rice University since 2009. He received a B.A. degree in philosophy from Yale University in 1973, and M.S. and Ph.D. degrees in computer science from New York University in 1980 and 1983, respectively. He was a Research Staff Member in the Software Technology Department at the IBM Thomas J. Watson Research Center from 1983 to 2009, and a manager there from 1987 to 2000. His research interests include programming models for high performance parallel and big data computing,
and static analysis-based tools for: programming language optimization; database applications; mobile security; compiling
for parallelism. He is an author or coauthor of more than 50 technical papers and 11 patents. Dr. Burke is an ACM
Distinguished Scientist and a recipient of the ACM SIGPLAN Distinguished Service Award. He is a member of IEEE
(Institute of Electrical and Electronics Engineers).

Salvatore Guarnieri IBM Software Group, P.O. Box 218, Yorktown Heights, New York 10598 (sguarni@us.ibm.com).
Mr. Guarnieri is a Software Engineer in the Security Division of IBM Software Group, and a Ph.D. student in the
Department of Computer Science of University of Washington. He is an author or co-author of six scientific publications.
His research interests include program analysis and security of scripting languages.

Marco Pistoia IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights,
New York 10598 (pistoia@us.ibm.com). Dr. Pistoia is a Manager and Research Staff Member at the IBM T. J. Watson
Research Center, where he leads the Mobile Technologies group. He has worked with IBM for seventeen years. He holds a
Ph.D. in Mathematics from New York University, Polytechnic Institute, and a Master of Science and Bachelor of Science
degree in Mathematics from the University of Rome, Tor Vergata, Italy. His research interests include program analysis,
programming languages, security, and mobile technologies. He is an author or co-author of ten books, thirty technical
papers, fourteen patents and sixty-eight patent applications, and the recipient of two ACM SIGSOFT Distinguished
Paper Awards.

Vivek Sarkar Department of Computer Science, Rice University 6100 Main Street, Houston, Texas, 77005
(vsarkar@rice.edu). Prof. Sarkar is a Professor of Computer Science and the E.D. Butcher Chair in Engineering at Rice
University. He conducts research in multiple aspects of parallel software including programming languages, program
analysis, compiler optimizations and runtimes for parallel and high performance computer systems. Prior to joining
Rice in July 2007, Vivek was Senior Manager of Programming Technologies at IBM Research. His past projects include
the X10 programming language, the Jikes Research Virtual Machine for the Java language, the ASTI optimizer used in
IBM’s XL Fortran product compilers, the PTRAN automatic parallelization system, and profile-directed partitioning and
scheduling of Sisal programs. Vivek became a member of the IBM Academy of Technology in 1995, was inducted as an
ACM Fellow in 2008. He holds a B.Tech. degree from the Indian Institute of Technology, Kanpur, an M.S. degree from
University of Wisconsin-Madison, and a Ph.D. from Stanford University.