Enhancing User Privacy on Android Mobile Devices via Permissions Removal

Quang Do[1] Ben Martini[2] Kim-Kwang Raymond Choo[3]
Information Assurance Research Group, University of South Australia
[1]doyqy002@mymail.unisa.edu.au; [2]ben.martini@unisa.edu.au; [3]raymond.choo@unisa.edu.au

Abstract

Android mobile devices are becoming a popular alternative to computers. The rise in the number of tasks performed on mobile devices means sensitive information is stored on the devices. Consequently, Android devices are a potential vector for criminal exploitation. Existing research on enhancing user privacy on Android devices can generally be classified as Android modifications. These solutions often require operating system modifications, which significantly reduce their potential.

This research proposes the use of permissions removal, wherein a reverse engineering process is used to remove an app's permission to a resource. The repackaged app will run on all devices the original app supported. Our findings that are based on a study of seven popular social networking apps for Android mobile devices indicate that the difficulty of permissions removal may vary between types of permissions and how well-integrated a permission is within an app.

1. Introduction

As mobile device usage increases in both ubiquity and capability, so will the need for increased security and privacy. Mobile devices are now being used for tasks once primarily undertaken on personal computers and notebooks. Paying bills, banking, ordering items online and others can now be done entirely on a smartphone. With the increase in the amount of sensitive information stored on a mobile device, user privacy becomes an important, if somewhat forgotten, factor. The most common operating system (OS) for mobile device, as of early 2013, is the Android OS by Google [1]. The Android platform is designed with openness in mind, meaning all of the system's source code is available for download, modification and review [24]. The Google Play Store uses a blacklist style of accepting Android applications ("apps"); that is all apps are accepted unless they are reported by users. Android relies on its permissions system in order to reduce the risk of a malicious app on a device. A user can manually check the list of permissions required by the app upon installation as a method to determine if it is a legitimate app. An app without the appropriate permissions cannot perform tasks requiring that resource. For example, a phone app requires the CALL_PHONE permission in order to make phone calls. By default, an app that is installed on an Android device can only be granted all of its requested permissions. While some resource permissions requested may have a legitimate use, others may be used for nefarious purposes.

It is a common finding in current research in Android privacy that the Android OS requires improvements in order to become a system that is capable of providing an adequate amount of user privacy. Third party frameworks that are built into customised versions of Android showcase what is possible through direct OS improvements. These improvements include enhancing the user friendliness of the current permissions system (e.g. permission categories). Another method of improving user privacy is the use of fine-grained permissions. This allows for users to allow or deny specific permissions on a per application basis. Other researchers attempt to lessen the impact of malware and over-privileged apps on a user's private and sensitive information by introducing the concept of shadow or mock data [4, 12, 13, 14]. Finally, a less discussed method of improving privacy is to reverse engineer and remove an app's permission, completely preventing the app's access to a resource. Research in Android privacy can be broadly categorised into permissions removal and fine grained permissions control, the former of which will be addressed in this paper.

This paper explores how Android social networking apps can be made more privacy friendly by permissions removal. The main focus of this paper will be on app permission removal using reverse engineering processes and we will discuss its viability in addressing user privacy concerns.

The layout of this paper is as follows: Section two presents the background information. Section three

978-1-4799-2504-9/14 $31.00 © 2014 IEEE
DOI 10.1109/HICSS.2014.623

discusses existing literature, while the following section explains the experiment setup and methods used to remove permissions from apps. Section five discusses the experiments' results. The last section concludes the paper and presents possible future work.

2. Android structure

Android apps are stored and distributed within an Android Application Package File (APK), a ZIP format file. Apps are commonly installed via the Google Play Store platform, which contains hundreds of thousands of apps created by third-party programmers and companies. Apps are generally unmoderated, and Google uses Google Bouncer [2], an in-house developed anti-malware application, to scan all submitted apps. The reliability of Google Bouncer is, however, questionable [3]. For example, Erturk [29] performed a case study on the prevalence of Android malware, and presents several forms of malware already common on the system.

The use of the Google Play Store allows automatic selection of appropriate app installation packages based on the device that is installing the app. For example, a tablet may require a higher resolution version of an app, containing tablet specific layouts.

2.1. APK File Structure

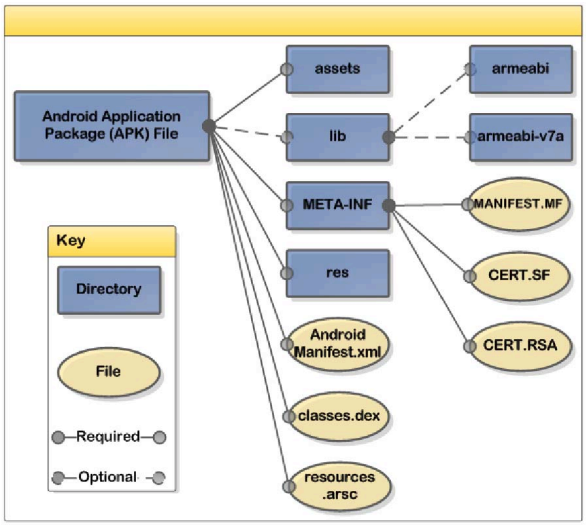

Figure 1. Overview of an APK file structure

An APK contains at a minimum, the directories and files shown in Figure 1. This AndroidManifest.xml file is most important in the research. This is stored in a binary XML format and must be converted to a plain text format before becoming human-readable. This file contains information such as the minimum Android version the app was designed for, the main activity (which is launched upon opening the app) and other details important to the basic functionality of an Android app. Most importantly for our purposes, it contains declarations of the Android permissions the app requires. Another file that will be used within this research is the classes.dex file, which contains the binary code of the app compiled to Dalvik byte code [4].

Programmers are free to add as many directories and files as needed to fulfill their requirements. Due to the inclusion of the manifest file detailing every file contained within an app, the structure is quite flexible.

Android apps are required to go through the application signing process before they can be installed onto a device. By default an Android system will not install an application if it is unsigned. This includes both physical and emulated Android systems. Generally for an organisation that releases Android apps, there is a single private key used to sign all their applications. By signing different applications with the same private key, they are able to share code and data as Android considers them to be within the same process [25].

2.2. Android Permissions System

Android uses a permissions-based approach to user privacy and security. Each app runs in its own virtual machine process, separate from all other apps currently running. Each Android app has a unique "Linux" User ID (POSIX). Two apps with different IDs cannot run in the same process [5]. This sandbox approach ensures that app data cannot leak to other apps.

Before installation of an app, a user is presented with a list of permissions the app requires. A user can only accept all permissions the app requires and install the app or cancel the installation completely. These permissions are defined by the AndroidManifest.xml file noted above, contained within the APK file in the root directory. An Android app's list of permissions is a reflection of the functionality of that particular app. A heavily over-privileged app (an app with too many permission requests) can act as a deterrent to users due to the long, potentially suspicious list of permissions requested. As of Android version 4.2.2, the Android OS has over 120 permissions [26]. Many of these permissions, though, have little effect on the privacy concerns of an Android smartphone user and are called normal permissions.

"Dangerous" permissions, on the other hand, are requested upon installation and explicitly defined in the AndroidManifest.xml file [27]. Figure 2 gives an

example of a dangerous permission; the highlighted row shows that the app requests access to the user's contacts.

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

Figure 2. Example of AndroidManifest.xml

3. Related Work

The Android OS is a mobile OS that is based on the use of sandboxing and an app permissions system wherein an app must first request controlled permissions from the system on installation. Many researchers working on Android security and privacy use the app permissions approach to improve user privacy on these devices. For example, Book, Pridgen & Wallach [6] examined a sample of 114,000 apps and found that the number of permissions required by apps are increasing, and consequently, posing a privacy risk to Android users. Shekhar, Dietz & Wallach [7] suggested that the additional increase of permissions may be due to the increasing popularity of in-app advertising, which requires the use of additional resources in order to cater for their own data collection, analysis and transmissions. A review of the current literature suggests that research on enhancing Android users' privacy in relation to apps permissions are broadly categorised into (1) improving the Android OS, (2) fine-grained app permission control, (3) mock data and (4) Android permission removal.

Studies by Felt et al. [8] and Kelley et al. [9] suggested that many users have a low comprehension of the Android permissions system – that is the permissions system may be insufficient in providing adequate user privacy in the hands of a novice user. Felt et al. then put forth several suggestions for improving the base Android OS, including showing the users risks of allowing certain permissions instead of just the resource and defining user-friendly categories for permissions. Kern & Sametinger [10] took a different approach and recommended the use of fine-grained individual permissions control on a per app basis. This means that each Android app would have each of their permissions explicitly listed and the user would either deny or allow the permission request. In their study, Kern & Sametinger also examined the use of extensions to the OS and third party apps in order to finely control an app's permissions, and developed their own app allowing a user to grant or deny a request as it occurs. In an independent yet related work, Zhou et al. [4] designed a system that could control an app's access to sensitive permissions – TISSA. With TISSA, the user can, for example, specify if the app is allowed to access the device's ID, contacts, call logs, etc. TISSA is even finer grained than the system proposed by Kern & Sametinger [10]. With TISSA, one could allow an app to access the device's contacts information but have the app receive faked or empty data (i.e. TISSA supports the use of mock and shadow data). Kern & Sametinger further found that to provide adequate control of app permissions, the apps would require repackaging specifically to reduce resource usage. Bugiel, Heuser & Sadeghi [11] instead presented some minor changes and improvements to the actual Android services located within the OS for fine-tuned control of app permissions. This differs from the previous research of Zhou et al. [4] and Kern & Sametinger [10] as a change in this area of the OS code could lead to the improvements being feasible in future versions or updates of Android. Most proposals for fine-tuned app control thus far require modification of the Android OS. With the use of a privacy control app as opposed to an OS modification, an app could possibly work on stock Android devices that have no OS changes.

The third area in improving Android users' privacy is that of the use of mock or shadow data. An example of this is sending simulated location data to apps that request it instead of the real location information. MockDroid [12] is a modified Android OS that allows the user to fake, to an app, the access or retrieval of a requested resource. One example use for this is an app that requires access to contact information in order to be installed on a device, but only requests the permission to data mine the device. A downside to this approach is that a complete wipe and installation of the modified Android OS is required to use MockDroid on a device due to the fact that it employs a custom Android system. Deploying this approach across many Android devices in an enterprise environment, for example, is thus not a feasible endeavour. AppFence [13] is another modified Android system aimed at imposing privacy controls on Android apps. When an app requests data that the user does not want it to be allowed, AppFence substitutes the data with fake "shadow data". For example, an app requesting a list of all contacts may get back an empty list whereas in reality, this is not the case. Shadow data can be used in almost all areas of the Android system, but once again, its use requires a modified Android OS. TaintDroid [14] is an approach to extending the Android OS that allows for detection of sensitive data leaving a device, as well as extremely fine-grained data access control. TaintDroid allows users to allow or deny apps from accessing data such as postal addresses, phone numbers, among others.

A less discussed type of Android app privacy is that of app permissions or resource removal. This approach requires an app be modified so that the permissions are

first selected and then removed. Generally this means an app's source code is required or the app is decompiled, modified to remove these permission requests and then recompiled. An unpublished manuscript [15] found that while it is possible to remove permissions manually from an app via the manifest file, it generally resulted in an app crashing or freezing at some point during its operation. Berthome et al. [16] proposed a set of two apps, comprising (1) the Security Monitor, a third party app installed onto the device, and (2) the Security Reporter, which would be injected into a decompiled target app. The injected app is able to monitor the targeted app and can then report to the Security Monitor with details such as resource requests. Juanru, Dawu & Yuhao [17] used a similar technique of decompiling Android apps to aid with their Android malware research. Xu, Saïdi & Anderson [18] developed a solution called Aurasium that automatically repackages Android apps to have sandboxing and policy enforcement abilities in order to enhance user privacy. They also identified, as in our research, that most research being done on Android privacy requires major modifications to the OS, resulting in usability issues. Permissions removal is a relatively new but promising approach as it does not require modifications to the Android OS.

4. Methodology

4.1 Experimental Setup

A total of seven Android apps were chosen to be examined and repackaged. A list of the apps selected along with the permissions they requested upon installation is shown in Table 1. The apps were chosen from the Social category of the Google Play Store based on the highest number of downloads and ratings at the time of research (March 2013). All the seven apps examined are free and are generally counterparts to popular websites and web services.

An X symbol in a cell in Table 1 represents the app from the top row requesting this permission within its AndroidManifest.xml file. Permissions which two or less apps requested have been omitted from the table for clarity.

In our experiment setup the apps were repackaged on a Windows 7 machine and tested on: a Samsung Galaxy i9000 (Android 4.1.1), i9100 (Android 4.2.2) and i9300 (Android 4.1.2). Only the i9000 and i9100 were rooted devices.

Permission	Facebook	Twitter	Instagram	Tango Text	Pinterest	LinkedIn	Tumblr
ACCESS_ FINE_LOCATION	X	X	X				
ACCESS_ NETWORK_STATE	X	X		X	X	X	X
AUTHENTICATE_ ACCOUNTS	X	X		X		X	X
CAMERA	X		X	X			
GET_ACCOUNTS	X	X		X	X	X	X
INTERNET	X	X	X	X	X	X	X
MANAGE_ ACCOUNTS	X	X		X		X	X
READ_CONTACTS	X	X	X	X		X	X
READ_ PHONE_STATE	X			X		X	
READ_ SYNC_SETTINGS	X	X		X		X	X
VIBRATE	X	X		X		X	
WAKE_LOCK	X	X	X	X	X	X	X
WRITE_CONTACTS	X	X		X		X	
WRITE_EXTERNAL_ STORAGE	X	X	X	X	X	X	X
WRITE_ SYNC_SETTINGS	X	X		X		X	X

Table 1. App Permission Requests

4.2 Permissions Selection

Before a permission request is to be removed, it must first be selected to be removed. When selecting a permission to remove or block, it must not affect the major functions of an app. For example, social networking apps require Internet access in order to function; as such the "INTERNET" permission is required. Testing an app without Internet access can be done simply by disabling all Internet connections. The aim, therefore, is to remove dangerous permissions from an app that should not be required. As such, the permissions that are most commonly requested by apps but also not necessarily required are considered for removal.

In order to determine what permissions are requested by each app, the app was first decompiled following the

process described in section 4.3. The AndroidManifest.xml file obtained can then be read with any plain-text editor. Table 1 displays the information obtained, and section 5.1 discusses the process through which we decided which permissions are reasonable to remove.

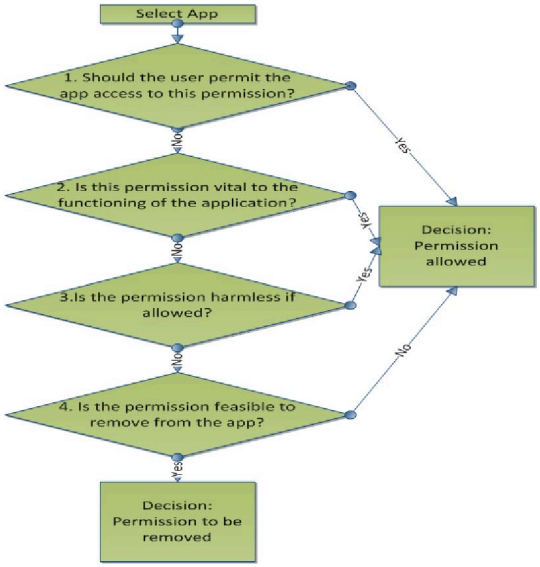

Figure 3. Permissions selection process

Figure 3 outlines our proposed app permissions selection process. The first step is for the user to determine whether the app requires this permission. The second step determines whether the app actually requires this permission in order to function. For example, a mapping app will require location resources such as the GPS system in order to function. A note keeping app, on the other hand, has no obvious need for such information. The next two steps will determine whether the permission is harmless and feasible to be removed from the app. For example, many app permissions allow an app to access sensitive information such as contact information, phone logs, IMEI numbers, and SMS. A user may choose to expressly disallow a particular permission even when the app has well defined justifications.

The feasibility of removing an app's permission is considered. Some apps may be so heavily integrated with a certain resource that it may not run without it.

4.3 Permissions Removal

In this paper, permissions removal is used in order to improve user privacy on Android devices. Permissions removal is the process wherein an app's package installer is reverse engineered to remove unnecessary or privacy-intruding permissions. The benefit of this method is that the app can be installed on any version of Android that supports the unmodified app. This means no additional third party software or rooted/custom Android OS is required which may have been an additional privacy/security risk.

A major downside to this method is the time required to properly remove one or more permissions and address dependencies within the app. It may not be possible to fully remove an Android permission's dependencies as the app's coded functionality may be too tightly integrated. For example, removing both coarse and fine locations from a turn-by-turn navigation app would not be useful or even viable due to the nature of the app. Another challenge with this method is that due to the digital signature verification in Android - the modified app is not signed with the original key and hence cannot be updated over the official version of the app installed on the device. This means a completely new installation of the app is required in order for this app to be updated on the (one) device.

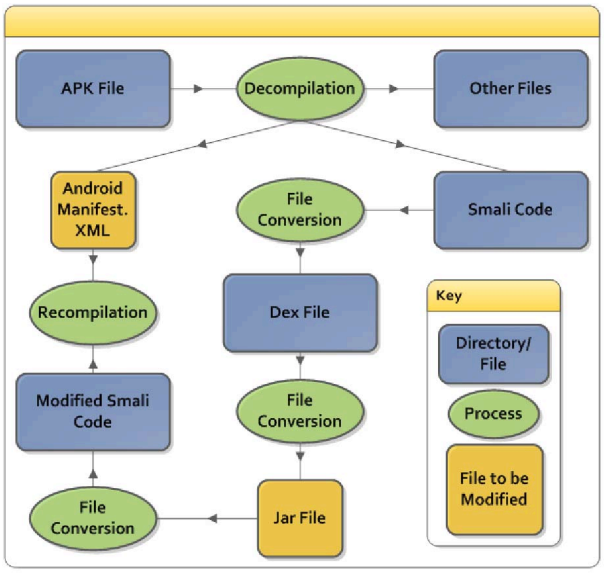

Figure 4. Ideal permissions removal process

Figure 4 shows the ideal method of manual permissions removal to be performed on an Android app. The reason this method is considered ideal is that this process results in the entire app's source code being readable and modifiable in Java. An app is first decompiled using a decompilation tool – in our case, APK Multi Tool is used [19]. Decompilation results in several files, as shown, with importance placed on the "smali" code files and AndroidManifest.xml file.

The smali code files are the source code of the particular Android app in a human readable format. The problem with this format is that it is difficult to read and

debug apps; the language is complicated and hard to understand. As this is the case, the smali code files are then converted to a single .dex or Dalvik Executable file using a tool called smali/baksmali [20]. This results in a .jar file, simply a Java archive file containing Java classes which can be read and extracted to .java files using JD-GUI [21]. At this point, changes to the app can be easily made by modifying its Java files.

The plain text AndroidManifest.xml file can now be read and modified using any plain text editor. Removing the highlighted row in Figure 2 would effectively render the app unable to read contacts data from the Android device, but may render the app unusable due to instability issues. Due to this, source code changes must be made in order to result in a usable app that cannot access contacts data.

After the source code changes are made, the app must be converted back into smali code in order for the recompilation process to be successful. The smali/baksmali software package is used once again to convert the Java code to smali code. APK Multi Tool is then used to recompile and sign the repackaged app. The result should be a working app installation package with some resource access removed, thus improving user privacy.

5. Results

5.1 Discussion: Android App Permissions

In our research, we found that the most commonly requested Android permissions were INTERNET, WAKE_LOCK and WRITE_EXTERNAL_STORAGE. These three permissions were requested by all seven apps examined in this research. Descriptions of what each Android permission entails are available on the Android Developers website [26].

An Internet connection is required to perform social networking tasks; as such the INTERNET permission is essential to the functioning of these apps. WAKE_LOCK allows the app to keep an app running in the foreground indefinitely without turning off the display [26]. This can result in excessive or unnecessary battery usage, but could be stopped simply by quitting the app.

WRITE_EXTERNAL_STORAGE allows an app to modify the contents of external storage devices such as attached USB drives and SD cards [26]. This also includes permission to write on the internal storage of a device, which is also defined as external storage. This permission is also essential to most app's functions. Due to these findings, the removal and blocking of the above three permissions will not be addressed as they are crucial to the inner workings of these apps.

The next most common among the apps examined were the ACCESS_NETWORK_STATE, GET_ACCOUNTS and READ_CONTACTS permissions, with only the Instagram app not requesting the permissions in each case.

ACCESS_NETWORK_STATE allows an app to detect whether Internet connectivity is currently available [26]. This is crucial to the app's user experience as it will allow the app to advise the user to enable an Internet connection if it is unavailable.

GET_ACCOUNTS is the permission that allows the app to retrieve the user account information from the phone [26]. This is also crucial to the smooth running of the app as the account manager contains information such as the user's email address, as well as accounts for specific services. For example, Facebook and Twitter accounts are stored in the Android account manager and accessed by an app with the GET_ACCOUNTS permission allowed. This speeds up the process of using these apps as the user does not need to manually login.

The READ_CONTACTS permission is, arguably, one of the most requested permissions thus far that apps should not really require or request in full. This further supports the fact that the existing permissions system lacks fine-grained permissions control. Facebook, for example, may retrieve contact names to help a user find other friends, but it may also acquire other unnecessary personal information. Contact information is of little use directly to most of the apps under examination as the apps should have their own user accounts with separate databases. Facebook users, for example, would have premade or would register for Facebook accounts. All their data would be stored in Facebook's databases, which would be accessible once they log into the app with their credentials. Because of this, the app should have no real need for phone contact data. Users' contact data may contain information that they do not want linked to their Facebook profiles such as contact addresses, full names, and Email addresses amongst other sensitive data. Instagram, Tumblr and Twitter especially have little to do with a user's contacts and should not have a need for requesting access to them. Although these services may require a contact's email address to perform some tasks, an app can only be granted all contact information or none.

Instagram and Tumblr are both primarily photo sharing apps. They have their own user accounts and databases containing submitted user information, and as such have no direct need for contact names and phone numbers, let alone other contact data. It is of questionable benefit to the user for the apps to be able to read this information. Data mining is a very distinct

possibility for the inclusion of these permissions in these apps [4]. Tango Text, Voice and Video (Tango Text) may be the only app that has a legitimate reason to request access to contact data as the app's purpose is to allow communication with contacts via text messaging, voice calls and video calls.

The AUTHENTICATE_ACCOUNTS, MANAGE_ACCOUNTS, READ_SYNC_SETTINGS and WRITE_SYNC_SETTINGS permissions are requested by all but two of the apps (Instagram and Pinterest). The AUTHENTICATE_ACCOUNTS and MANAGE_ACCOUNTS permissions allow an app to manage their user's credentials which are stored within the Android AccountManager service. The combination of these two permissions gives an app the ability to add their users' accounts to the Android system, allowing users to quickly authenticate their accounts and passwords [22].

Similarly, READ_SYNC_SETTINGS and WRITE_SYNC_SETTINGS are used by the apps to check whether the app can be synced and to change this setting respectively [26]. This setting can be managed manually by the user within the Android settings page via the list of third party accounts stored within the device.

The VIBRATE and WRITE_CONTACTS permissions are requested by four of the seven apps tested – Facebook, Twitter, Tango Text and LinkedIn.

Requesting the WRITE_CONTACTS permission allows an app to write to a user's contact data but not read this data [26]. This differs from the READ_CONTACTS permission in that the app is not allowed to access contact data if it requests only this permission. A combination of these two permissions is required for an app to modify or remove a current contact's data. An app that requests the READ_CONTACTS permission without the WRITE_CONTACTS permission is merely trying to retrieve a user's contact information, without this act of data gathering being of any benefit to the user.

Finally, ACCESS_FINE_LOCATION and READ_PHONE_STATE are requested by three of the seven examined apps – see Table 1.

The Android OS has two methods to provide an app with location information. A general location can be obtained by accessing cell tower and Wi-Fi information. This can be accessed by an app that has requested the ACCESS_COARSE_LOCATION permission [26]. A more accurate location can be obtained when an app requests the ACCESS_FINE_LOCATION [26]. This uses the inbuilt GPS hardware to derive very accurate location information. Facebook, Twitter and Instagram are the three apps that request this permission. Location information is another questionable permission when requested by apps that are not intrinsically GPS apps or mapping apps. Studies by Zhou et al. [4] found that apps do leak location data. Therefore, this permission should be removable from apps without affecting functionality while enhancing privacy.

READ_PHONE_STATE, a seemingly innocuous permission, is actually capable of providing the app with a large amount of phone information. It allows the app to access the phone number of the device, the IMEI (International Mobile Equipment Identifier) of the device and whether a call is active, as well as the caller number [26]. As an IMEI uniquely identifies a device and is often used to locate a device, the leakage of this information can be detrimental to the user. An IMEI can also be used to blacklist a device from mobile networks by calling a network provider and supplying them with the IMEI. The only reason a social networking app would require this permission is to check whether a call is active in order for the app to act appropriately, be it pause the current processing or save a draft if the user were in the process of writing a message. Thus this permission would be a good candidate for removal.

In summary, the permissions we have determined are a common privacy risk to the user are: READ_CONTACTS, ACCESS_FINE_LOCATION, and READ_PHONE_STATE.

5.2. Repackaging Android Apps

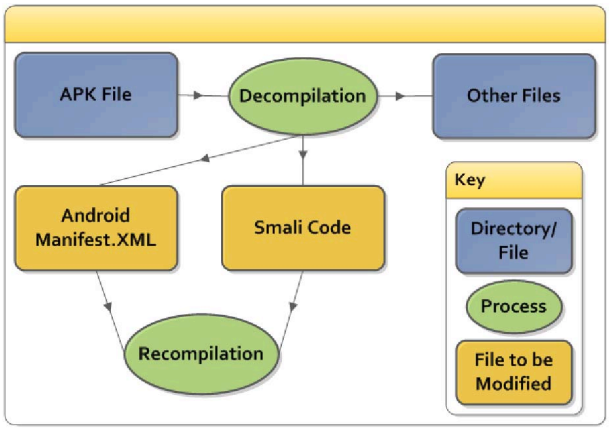

Figure 5. Alternative removal process

Following the steps presented in Figure 4, we encountered a major problem. The current process for converting smali code to Java code is not yet fully functional. The resulting Java code, upon conversion, contains many thousands of errors. Another reason this approach may not be so ideal is that the most commonly used software package to convert the smali code into Java files, DEX2JAR [31], has a known backdoor that

could be exploited [30]. Therefore, an alternative approach must be undertaken in order to ensure a working app at the end of the decompilation and recompilation process – see Figure 5.

The approach outlined in Figure 5 is capable of recompiling to an error-free app as the smali code is a direct representation of the app.

We decided to use Virtuous Ten Studio due to its ability to decompile and recompile apps without the need for an external tool as well as accommodate the editing of smali and xml files. The program also includes an inbuilt documentation system that allows for the syntax highlighting of smali code files.

5.2.1 ACCESS_FINE_LOCATION

Of the apps tested in this research, three requested this Android permission – Facebook, Twitter and Instagram. For social networking apps, location information is not crucial to the functionality of the app. As such, it is a simple matter of modifying the AndroidManifest.xml file of these apps to exclude the location permission in order to prevent the app from requesting any location information.

Removing this permission from each of the apps resulted in a fully usable app, apart from their inability to obtain user location. There were no crashes or force closes of the apps, although as previously mentioned, the app cannot be installed over a currently installed official version of the app due to the use of a different key in the app signing process. A removal and reinstall of the modified app is required, meaning all app data is lost. The permission can be removed without usability issues, which is due to dependencies within the app. This is because it is similar to running in flight mode, with Internet access enabled. Removing this permission manually simply makes the app think this is the case. It is clear that this resource is not crucial to the functioning of the Social Networking apps tested.

5.2.2 READ_CONTACTS

This permission is requested by Facebook, Twitter, Instagram, Tango Text, LinkedIn and Tumblr. The READ_CONTACTS permission is one that is generally very tightly integrated into the functioning of a Social Networking app. Although these are social networking apps, it does not necessarily mean that they should be given full access to all contact information on the device.

Unlike the ACCESS_FINE_LOCATION permission, READ_CONTACTS cannot simply be removed from an app with no adverse effects. When this permission is removed from the Tumblr app's AndroidManifest.xml file, the main functions of the app remain fully functional. However, upon importing contacts, the app remains in an infinite looping state until the app is closed.

Contact data in an Android system is stored in the form of a SQLite database. An app cannot access this database directly unless it has root permissions. Otherwise, it must first request the READ_CONTACTS permission in its manifest file and then import the ContactsContract package from the Android libraries. This package allows the app to read in data by querying the database via API calls. As this is the main method used by Android programmers to access contact data, in order to remove an app's dependencies with contact data, this would be a logical starting point.

Contact data is imported into Tumblr in the ImportContactsTask class section of its code. This code simply accesses the contacts database on the device and retrieves the email information of each of the contacts. In order to find the ImportContactsTask class within the large amount of source code within the Tumblr app, the ContactsContract package was used as the basis of a keyword in order to search through these files. This code is called only when a user accesses the "Find blogs to follow" page and clicks on their own contacts. Tumblr advises the user that their contact data is being read and searched. The contact's email is used by Tumblr to determine if the contact has a Tumblr blog that the user may follow. By only removing the READ_CONTACTS permission from the manifest file, the search for blogs to follow becomes never-ending; the app continuously searches for contacts that it will never have access to. This is non-functional behaviour and therefore must be changed in the source code. By removing the READ_CONTACTS permission from Tumblr's manifest file and modifying the ImportContactsTask class to no longer read in the Android device's contact information, the Tumblr app is no longer able to request contact information. In order to prevent Tumblr from attempting to read contact data, two methods are possible. One method is to completely disable the button, which causes the app to read in the contacts. Another method is to simply return an empty list as the list of contacts. Both of these methods were attempted on Tumblr and were successful.

Upon removal of the READ_CONTACTS permission from the AndroidManifest.xml file, Facebook is able to launch and the user is able to perform all tasks except for finding friends. Performing this task results in an unexpected close of the app. A similar technique to the one used to make Tumblr usable can be used here to prevent the user from searching for friends.

Removing the READ_CONTACTS permission from its AndroidManifest.xml will result in Instagram finding zero contact on a device that has contacts. The rest of the functionality of the app is unaffected. This is the ideal outcome for the permissions removal process.

When removing the READ_CONTACTS permission from Twitter's AndroidManifest.xml list of permissions, the app is not adversely affected until one starts to import contacts into Twitter. Upon importing contacts into Twitter on a device with existing contacts, Twitter closes unexpectedly. This suggests the app may use contact data with good intentions. Once again, the method used to make Tumblr and Facebook usable could be used here.

Removing the READ_CONTACTS permission from the LinkedIn app causes the app to force close when a user enters the settings screen. This makes the app nearly non-functional and must be resolved.

The Tango Text app has justification for the READ_CONTACTS permission as it is also used as a phone call and text messaging replacement app and so we did not attempt to remove its permission in our research.

5.2.3. READ_PHONE_STATE

Facebook, LinkedIn and Tango Text all require the READ_PHONE_STATE permission in order to be installed. This permission allows the app to access several important resources including: turning off an app's sound when there is an incoming call, accessing the phone's IMEI information and other personal information [32].

Removing this permission from both Facebook and LinkedIn caused no obvious side effects. This may be because the apps only use this permission in order to mute themselves upon an incoming phone call. READ_PHONE_STATE also gives an app the 64-bit hexadecimal ID, which is unique to the phone and is sometimes used instead of a user login [28]. Apps that rely on this ID may become inoperable.

Tango Text is one app that makes use of these unique device IDs. Removing this permission from its manifest file causes an "error 62" message to appear, and renders the app unusable. Upon removing instances where Tango Text tries to read in the phone's unique identifier, the Tango Text app crashes upon startup, due to its empty or corrupt device ID.

6. Conclusion and Future Work

From the apps that were tested in this research, it was found that ACCESS_FINE_LOCATION could be removed from social networking apps without requiring direct source code changes. The only change required was to actually remove the permission request from the AndroidManifest.xml file and then recompile the app. The apps, with the location permissions removed, were functionally stable after the permission removal. Apps such as Facebook, which have use of the location request in some of their features are, of course, impaired in this aspect.

To prevent apps from reading contacts information generally requires some code changes in order to prevent the app from crashing or stalling. Most of the apps, even though they were social networking applications, have a process that a user must go through before the app can read the contacts information. This is generally an option labeled in a similar vein to "Import Contacts" and implies that the app has been programmed to be user privacy friendly. The one exception to this, in our research, was the LinkedIn app, which would attempt to read contact data upon opening of the settings screen.

From these results, it can be determined that it is indeed possible to remove permission requests from apps via reverse engineering and result in a usable and privacy friendly app.

Removing the READ_PHONE_STATE permission from Facebook and LinkedIn had a similar result to removing location access from them. With Tango Text, which required this permission in order to identify the phone, the app would instead display an error message. This means that successful removal of this permission would depend on the app in question.

All the research undertaken in this paper on permissions removal was manually completed, therefore future research recommendations include automating this process. By using heuristic methods, it may be possible to locate sections within the source code where permissions resources are used automatically. The decompilation, modification and recompilation process could then be completely autonomous.

This leads onto a further future work or research that could be undertaken where this automated system would be implemented onto an Android device. This would result in a self-functioning system that could enhance the privacy of apps on the device.

References

[1] M. Oleaga, 2013, "OS vs. Android Market Share 2013: Google Mobile Platform Dominating Apple Worldwide in March Figures", http://www.latinospost.com/articles/15039/20130322/ios-vs-android-market-share-2013-google-mobile-platform-dominating.htm, accessed 25 March 2013.

[2] O. Hou, 2012, “A Look at Google Bouncer”, http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/, accessed 14 April 2013

[3] FY. Rashid, 2012, “Researchers upload dangerous app to Google Play store”, http://www.scmagazine.com.au/News/310192,blackhat-researchers-upload-dangerous-app-to-google-play-store.aspx, accessed 14 April 2013

[4] Y. Zhou, X. Zhang, X. Jiang & V. Freeh, “Taming information-stealing smartphone applications (on Android)”, TRUST 2011, pp. 93-107.

[5] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev & C. Glezer, “Google Android: A Comprehensive Security Assessment”, IEEE Security & Privacy Magazine, vol. 8, no. 2, 2012, pp. 35-44.

[6] T. Book, A. Pridgen & DS. Wallach, “Longitudinal Analysis of Android Ad Library Permissions”, arXiv preprint arXiv:1303.0857, 2013.

[7] S. Shekhar, M. Dietz & D.S. Wallach, “Adsplit: Separating smartphone advertising from application”, CoRR, abs/1202.4030, 2013.

[8] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin & D. Wagner, “Android permissions: User attention, comprehension, and behavior”, SOUPS 2012, p. 3.

[9] P.G. Kelley, S. Consolvo, L.F. Cranor, J. Jung, N. Sadeh & D. Wetherall, “A Conundrum of Permissions: Installing Applications on an Android Smartphone”, Financial Cryptography and Data Security 2012, pp. 68-79.

[10] M. Kern, & J. Sametinger, “Permission Tracking in Android”, UBICOMM 2012, pp. 148-155.

[11] S. Bugiel, S. Heuser & AR. Sadeghi, “myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android”, Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt (CASED), 2012.

[12] AR. Beresford, A. Rice, N. Skehin & R. Sohan, “MockDroid: trading privacy for application functionality on smartphones”, HotMobile 2011, pp. 49-54.

[13] P. Hornyack, S. Han, J. Jung, S. Schechter & D. Wetherall, “These aren't the droids you're looking for: retrofitting android to protect data from imperious applications”, ACM CCS 2011, pp. 639-652.

[14] W. Enck, P. Gilbert, BG. Chun, L.P. Cox, J. Jung, P. McDaniel & AN. Sheth, “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones”, 9th USENIX conference on Operating systems design and implementation, 2012, pp. 1-6.

[15] J. Helfer & T. Lin, 2012, “Giving the User Control over Android Permissions”, http://css.csail.mit.edu/6.858/2012/projects/helfer-ty12.pdf, accessed 25 March 2013.

[16] P. Berthome, T. Fecherolle, N. Guilloteau & JF. Lalande, “Repackaging Android Applications for Auditing Access to Private Data”, ARES 2012, pp. 388-396.

[17] L. Juanru, G. Dawu & L. Yuhao, “Android Malware Forensics: Reconstruction of Malicious Events”, ICDCSW 2012, pp. 552-558.

[18] R. Xu, H. Saïdi & R. Anderso, 'Aurasium: Practical policy enforcement for android applications', 21st USENIX conference on Security symposium, 2012, pp. 27-27.

[19] http://apkmultitool.com/, accessed 11 April 2013

[20] http://code.google.com/p/smali/, accessed 11 April 2013

[21] http://java.decompiler.free.fr/?q=jdgui, accessed 11 April 2013

[22] C. Mann & A Starostin, “A framework for static detection of privacy leaks in android applications”, ACM SAC 2012, pp. 1457-1462.

[23] http://www.virtuous-ten-studio.com/, accessed 4 May 2013

[24] http://source.android.com/, accessed 13 June 2013

[25] S. Bugiel, S. Heuser & AR. Sadeghi, *myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android*, Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt, 2012.

[26] http://developer.android.com/reference/android/Manifest.permission.html, accessed 14 June 2013

[27] http://developer.android.com/guide/topics/manifest/permission-element.html, accessed 13 June 2013

[28] http://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID, accessed 13 June 2013

[29] E. Erturk, 'A case study in open source software security and privacy: Android adware', WorldCIS 2012, pp. 189-191.

[30] R. Unuchek, 2013, “The most sophisticated Android Trojan”, http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan, accessed 13 June 2013

[31] http://code.google.com/p/dex2jar/, accessed 13 June 2013

[32] http://developer.android.com/reference/android/telephony/TelephonyManager.html#getDeviceId(), accessed 14 June 2013