An empirical study of collaborative model and its security risk in Android

Ajay Kumar Jha and Woo Jin Lee
School of Computer Science and Engineering, Kyungpook National University
Daegu, Republic of Korea
ajaykjha123@yahoo.com, woojin@knu.ac.kr

Abstract: Android provides a framework for the development of collaborative applications, which is considered as one of the reasons behind its success. Collaborative model provides flexibility to an application in utilizing services offered by other applications. This approach offers several advantages to developers, such as allowing them to dedicate all of their resources in developing only core functionalities of an application while leveraging services offered by other applications for its auxiliary functionalities. However, the collaborative model also has some disadvantages, such as opening of attack surfaces in an application during exposure of some of its components as it offers its services. Malicious actions can be performed through the exposed components of the application. Android provides permission-based security to protect the exposed components. However, developers must implement the security correctly. In this paper, we empirically evaluate the scale of the collaborative model adopted by Android applications. We also investigate various methods to achieve collaboration among applications. Furthermore, we evaluate the scale of security risk instigated by the collaborative model and perform several other empirical studies on 13,944 Android applications.

Keywords: Android applications, inter-application communications, collaborative application model, permission-based security, security risk assessment

1. Introduction

Android operating system, which currently holds the largest market share [1], is one of the major players in the smartphone industry. Although open source is considered as the main reason behind its remarkable success, it can also be attributed to the incentives available for each stakeholder including developers. Developers can develop Android applications on multiple platforms. Most importantly, they can utilize the collaborative framework during application development. Android application developers can also utilize large number of APIs provided by the platform. These incentives not only facilitate but also encourage an individual developer to develop and publish applications in the market, which is evident as Google Play Store has overtaken other application stores based on the number of available applications, although it generates less revenue for developers than its main rival, the iOS App Store [2]. The framework for developing collaborative applications is a unique feature of Android and it is considered as one of the key elements behind Android success.

Collaborative model is not a new approach. Android only brought this approach into application development practice, which cannot be considered as a trivial step, but rather as a paradigm shift in the way applications collaborate. Some forms of collaboration exist among traditional software. For example, plug-ins are developed to collaborate with the target software. Android brought this traditional approach to a new height. Unlike traditional software where plug-ins act only as supplementary, an Android application that offers services to other applications can also perform its own tasks. For example, a camera application performs its own tasks as well as offers services to other applications. Other applications can easily utilize the services offered by the camera application. This seamless collaboration between applications is a unique feature of Android. The collaborative approach provides several advantages to developers, such as allowing them to focus on an application's core functionalities while leveraging services offered

by other applications for its accessory functionalities, which increases the quality of the application and decreases the development time, resulting in less development cost and competitive advantage in the market.

Along with several advantages, the collaborative approach has some disadvantages. Android applications are composed of components. An application offers its services by exposing one or more of its components, which opens attack surfaces [8, 11]. These attack surfaces can be easily exploited by malicious applications. Android provides protection against these attack surfaces at various places. Applications are vetted for malicious behavior when they are installed into Google Play Store [27]. The procedure prohibits malicious applications from entering the store and collaborating with benign applications. Given that the vetting procedure is not 100% effective [28], some malicious applications may enter into the store. The presence of malicious applications in the store has been reported occasionally [29, 30]. Third party stores are other but major source of malicious applications [31] against which Google provides a verification process [32]. However, the process is optional for users. Android also provides permission-based security at the application level to protect the attack surfaces. However, developers must implement the security correctly. Several studies have indicated that developers fail to implement the collaborative model and its security correctly. A common developer's mistake in Android is to expose a component unintentionally [3], which leaves it unprotected. Developers may fail to implement protection mechanisms correctly because most of the developers are not security experts [4, 5].

In this paper, we empirically evaluate the scale of the collaborative model adopted by Android applications. The evaluation results can shed light on some important points. If the collaborative model has been adopted in large scale, then it can be concluded that the collaborative model has succeeded and it is contributing to the success of Android. The success of the collaborative model in Android may also open doors for collaborative applications or software on other platforms. Meanwhile, small-scale adoption indicates that the collaborative model has failed and it does not contribute to the success of Android. We also empirically evaluate the security risk instigated by the collaborative model. If the collaborative model has been adopted in large scale and the empirical study indicates that most of the exposed components are protected, then it can be concluded that the collaborative model does not pose significant security risk. Meanwhile, a large-scale collaborative model with a large number of unprotected exposed components indicates that either the developers are unaware of the security risk associated with the exposed components or they are implementing the security incorrectly. In either case, the collaborative model poses high security risk, which may affect its further development in Android or other platforms. Furthermore, we investigate the common security, reliability, and availability issues associated with the collaborative model and perform empirical studies on 13,944 free Android applications downloaded from the Google Play Store.

The main objective of this study is to investigate the collaboration among Android applications and the security risk instigated by the collaborative approach. In this direction, the contributions of this paper can be summarized as follows:

- Performs empirical studies on 13,944 popular Android applications.
- Assesses the scale of collaboration among Android applications and reports on different collaboration approaches involving the participation of different types of components.
- Evaluates the security risk in the collaborative approach by analyzing unprotected components and sensitive resources used by the applications.
- Identifies incorrect implementations of the collaborative model and its security and discusses their implications.

The rest of the paper is organized as follows. Section 2 describes the background on the collaborative model in Android and its security. Related works are discussed in Section 3. Several empirical studies converging toward the collaborative model and its security risk are performed in Section 4. We also discussed the incorrect implementations of the collaborative model and its security along with their implications in Section 4. The results obtained from the empirical studies are discussed in Section 5. Threats to the correctness of the obtained empirical results are discussed in Section 6. Finally, the paper concludes in Section 7.

2. Collaborative model and its security

Android applications comprise four kinds of components: activity, service, broadcast receiver, and content provider. In an application, a component can complete a task independently. However, components may interact with each other to complete the task. Furthermore, the components of an application can also communicate with the components of other applications. For example, a user may need to attach an image in an email. In this case, an email application may communicate with a camera application to obtain the image. This kind of collaboration among applications is a unique feature of Android, which can be described in terms of a client-server architecture. In a collaboration event, an application that offers services acts as a server whereas an application that avails the services acts as a client. Unlike the client-server architecture, Android applications are neither clients nor servers exclusively. In the example, the email application acts as a client whereas the camera application acts as a server for the event. However, the condition may reverse in another event. For example, a user captures an image through a camera application, and then sends the image through an email application. In this collaboration, the camera application acts as a client whereas the email application acts as a server.

The collaboration in Android is achieved through intents and intent filters [6]. An intent describes an operation to be performed by a component. Depending on the presence or absence of a target component, an intent is categorized as either explicit or implicit. An explicit intent is directly addressed by a target component whereas a target component for an implicit intent is resolved by the system. Intent filters play a crucial role in the target resolution process. A component advertises its capability to handle a specific operation through an intent filter, which is registered in the system. If an implicit intent requests for a target component, then the system matches the operation to be performed by the intent with the operation advertised by the components. The component with the matching operation is served as a target component for the intent. Although collaboration can be achieved through an explicit intent, it requires prior knowledge of the target application, which results in some restrictions on the collaboration. Meanwhile, the collaboration achieved through an implicit intent does not impose these restrictions. In this form of collaboration, the collaborating applications do not have knowledge of each other. An application advertises its services through intent filters, and any other application can avail those services if they satisfy the security condition imposed by the service provider application.

The collaborative model and its security are implemented in Android through a manifest file, which is a configuration file where all components of an application must be declared. However, the broadcast receiver components can also be declared in the source code. An application willing to offer services to other applications must be declared through the manifest file. Given that services are offered through one or more components of an application, the application must expose its components. A component of an application makes itself available to other applications by setting its exported flag attribute to true. In the case of activity, service, and broadcast receiver components, the default value of the exported flag depends on an intent filter. The presence of one or more intent filters in a component indicates that the component is exported. In the case of a content provider component, if an application uses an API level less than 17, then the content provider component is exported by default.

Security in Android is implemented at both system and application levels. At the system level, Android uses the sandboxing technique, which creates a virtual isolated execution environment for an application. However, the isolation breaks when an application communicates with another application. Here, Android provides permission-based protection at the application level. Permissions are used to protect the exposed components, which participate in inter-application communications. A permission can be compared to a label, which is attached to a sensitive object. A subject willing to access the sensitive object must first acquire the permission. An application that offers services to other applications declares a permission and then protects its exposed component with the permission. Other applications willing to avail the services must use or acquire the permission declared by the application containing the protected component. The acquired permission must be accepted by users during application installation. Since Android 6.0 or API level 23, users have to accept the permission during runtime. Additional details on permission-based security in Android can be found in [8, 9, 34].

We illustrate the collaborative model and its security in Android through a motivational example shown in Figure 1, which presents three applications: A_1, A_2, and A_3. Applications A_1 and A_2 offer services to other applications by exposing some of their components. Application A_1 does not protect its exposed component C_{12}. Application A_2 protects one of its exposed component C_{21} with permission P while leaving another exposed component C_{23} unprotected. Components C_{12} and C_{23} of applications A_1 and A_2 can be accessed by any application without any

restrictions while component C_{21} of application A_2 can be accessed only by applications that have already acquired the permission P. As shown in the figure, application A_1 has acquired the permission P; therefore, all components of A_1 can access the protected component C_{21} of application A_2. Let us assume that application A_3 is a malicious application, which tries to access the component C_{21} but fails because the component is protected and it has not acquired the permission P. However, application A_3 can easily access the component C_{23} of the same application because it is not protected with a permission. Although application A_3 cannot directly access the component C_{21} of application A_2, it can access that component indirectly. Application A_3 uses the exposed unprotected component C_{12} of application A_1 to access the component C_{21}. This process is possible because the exposed component C_{12} is not protected by a permission and it has acquired the permission to access the component C_{21}. This type of attack is called privilege escalation attack or confused deputy attack [4, 5, 7].

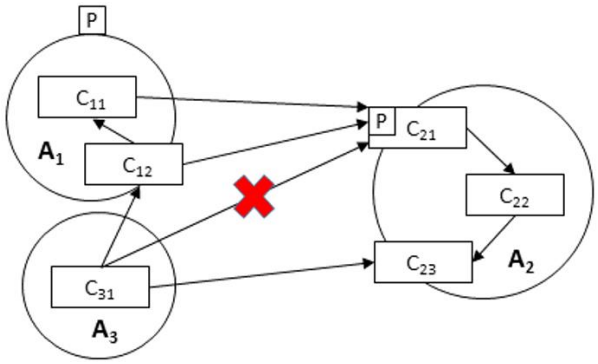


Figure 1. A motivational example of the collaborative model and its security.

The example shows that protecting some of the exposed components does not protect an application. A developer should protect all exposed components of an application. Despite having all exposed components of an application protected, a malicious application can attack the application through other unprotected applications. Thus, the collaborative model can be risky even for the protected applications if some of the applications installed on the device are unprotected.

3. Related works

Recently, security and privacy issues in Android have been extensively investigated. Many tools and techniques have been proposed for identifying security issues in Android applications including the security issues related to the collaborative model. ComDroid [11] performs analysis of inter-application communication vulnerabilities on 100 Android applications. It statically analyzes the components and intents of an application. If a component has been exposed to other applications and it is either not protected by a permission or protected by a weak permission, then a warning is generated. ComDroid also generates a warning when an application tries to communicate with another application through an implicit intent with no permission or a weak permission. It defines weak permissions as those permissions that are declared with either normal or dangerous protection level. Enck et al. [12] performed a source code analysis on 1,100 Android applications. They investigated the security and privacy issues of inter-application communications caused by implicit intents and unprotected dynamic broadcast receivers including several security and privacy issues of Android applications. However, they did not perform analysis on the manifest file, which contains most of the information on the exposed components. EPICC [13] mapped the inter-component communications both within an application and among applications. It determined the entry and exit points of applications by analyzing the values of intent filters and intents, respectively. In the process, EPICC also analyzes the exposed components. Finally, it connects the exit points with possible entry points. Furthermore, it also performs analysis of inter-component communication vulnerabilities similar to ComDroid on 1,200 Android applications.

Many researchers have investigated possible attacks attributed to exposed components, in addition to general inter-application communication vulnerabilities. One widely investigated attack is the privilege escalation attack. Felt et al. [5] examined 872 Android applications for privilege escalation vulnerability. They found that 320 out of 872 applications access sensitive resources protected by permissions and at least one service or broadcast receiver

component is exposed but unprotected in all 320 applications. They also proposed an IPC inspection technique as a defense against privilege escalation attack. Covert [14] mainly analyzed the privilege escalation vulnerability along with several other inter-application communication vulnerabilities in 500 Android applications. It extracts essential information from an application through static analysis. The extracted information from different applications is then combined and converted to a formal specification language. It then performs compositional analysis to find the vulnerabilities. CHEX [15] analyzed 5,486 Android applications for component hijacking vulnerability, which includes all kinds of attacks that gain unauthorized access to protected or private resources through exposed components. It is not only limited to privilege escalation or confused deputy attack, but also includes those attacks where permission protection is not explicitly involved.

Developers commit mistakes of exposing components or intents even for intra-application communications, which increase the attack surface. ComDroid [11] recommends changes in the Android platform to reduce the attack surface. Kantola et al. [3] implemented a heuristic-based approach in the Android platform to make the exposed components and intents private, which reduces the attack surface. Felt et al. [22], Au et al. [23], and Bartel et al. [24] separately investigated applications that use more permissions than required. These over-privileged applications can become accessories in a privilege escalation attack.

Researchers have performed empirical studies on permission-based security as well as inter-application communications in Android. Li et al. [16] performed a very short empirical study on 2,000 Android applications for permission-protected components. They found a negligible number of components protected in comparison to the number of components exposed. Barrera et al. [21] studied permission use pattern on 1,100 applications. They used the self-organizing map to visualize the permission use pattern based on the category of applications. Maji et al. [25] and Sasnauskas and Regehr [26] in separate works investigated the robustness of both intra-application and inter-application communications. They discovered that components are vulnerable against receiving unexpected intents, which cause them to crash.

In contrast to existing related works, our work does not focus on identifying or solving any specific security issue, such as privilege escalation attack [5, 14, 15], over-privileged applications [22, 23, 24], or component leaks [16]. Our work concentrates on evaluating the security risk caused by the collaborative model through a large-scale empirical study. However, similar to some existing works [3, 11, 21], we identify and discuss the incorrect implementations of the collaborative model and its security. Furthermore, we discuss some of the security issues, such as issues related to custom permissions, which have never been discussed previously in detail.

4. Empirical studies

We performed empirical studies on 13,944 free Android applications downloaded from Google Play Store in May-June 2015. The top 500 free applications from each category displayed in the Play Store were downloaded manually by one of the authors. In some categories, there were less than 500 free top applications. Many applications were placed in multiple categories. Only one instance of these applications was downloaded. Google Play Store is localized. We used the Tor Browser [17] to localize the Play Store to the United States when downloading the applications. Android applications are distributed in APK format. We used the Apktool [18] for reverse engineering of APK files of each downloaded Android application and extracted their manifest file. We then performed analysis on those manifest files. To perform analysis, we wrote a small program, which parses the manifest file and extracts the required information. In many cases, especially those cases where incorrect implementations were detected, the extracted information were manually verified by one of the authors. Among the 13,944 applications in the dataset, there are 461 duplicate applications with different versions. In this empirical study, we have treated the duplicate applications with different versions as separate applications because the manifest files are modified in many of those applications. Although the collaborative model and its security risk is the main target of our study, we also performed several other empirical studies.

4.1 Size of Android applications

Size is considered as an important metric in the software industry. Most commonly, it is measured in number of lines of code (LOC). Android applications are composed of components, which have pre-defined lifecycle. These components are created under the skeleton of their life cycle methods. Android programmers also make extensive use

of the API provided by the framework. Under these circumstances, the size of a component is bound to a certain length. Instead of measuring the size of an Android application in terms of LOC, we define the size of an application in a higher level of abstraction, which is the number of components (NOC) in an application. We classified applications into small, medium, and large size depending on the number of components up to 10, 11 to 50, and above 50, respectively.

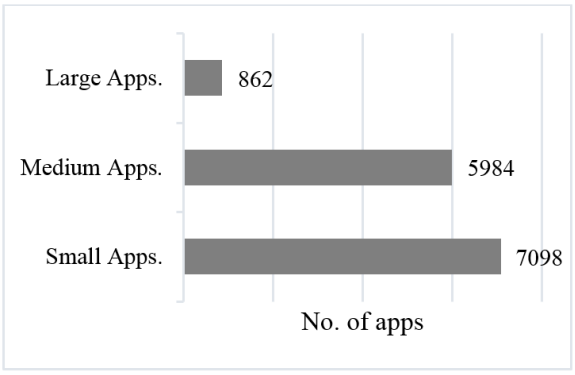


Figure 2. Application size in the dataset.

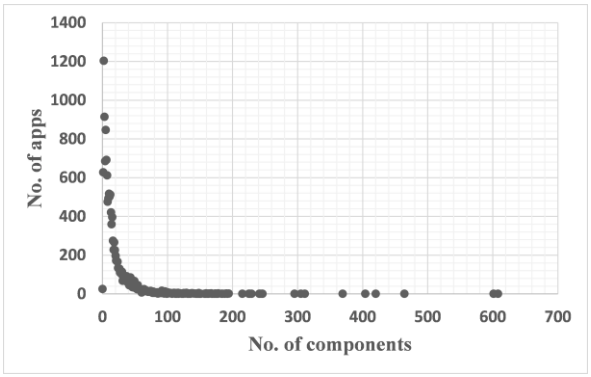


Figure 3. Number of components in applications.

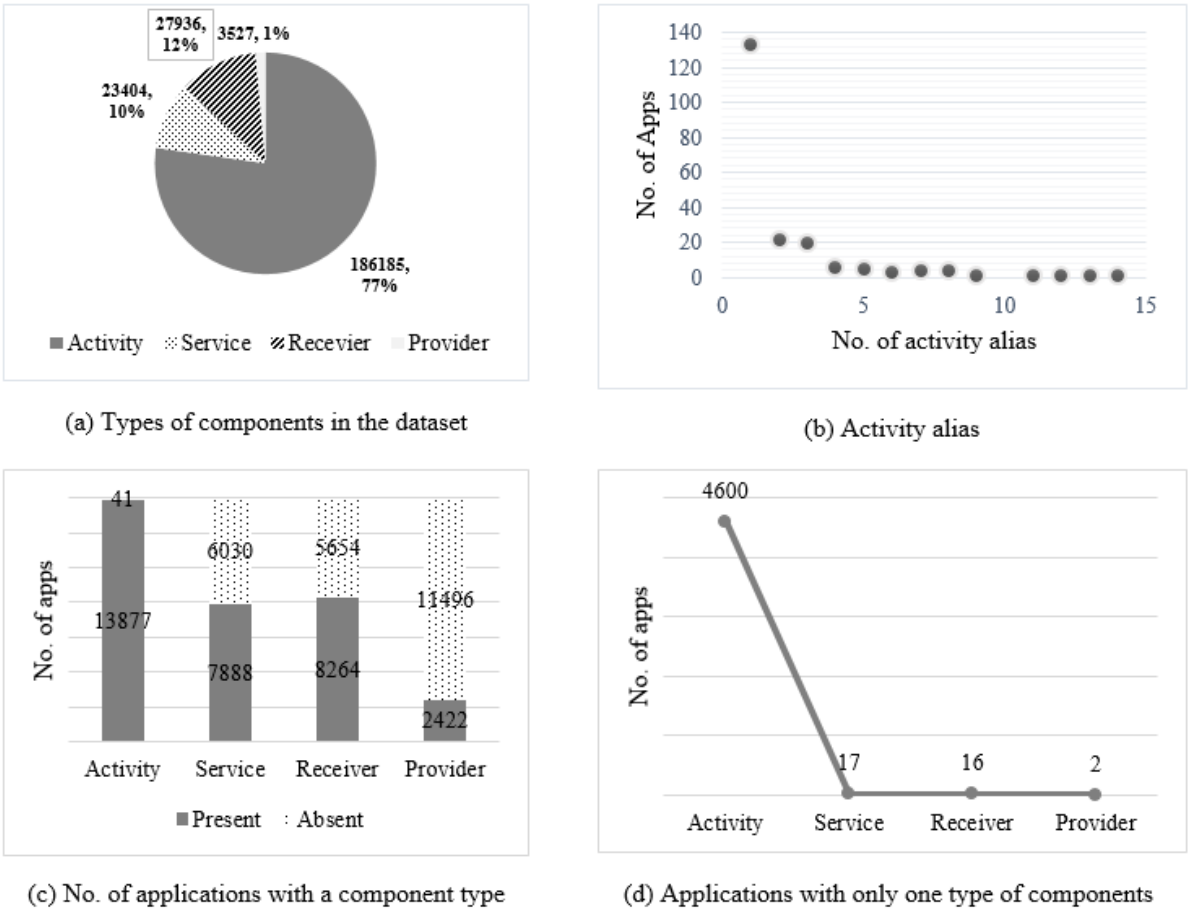
In the dataset of 13,944 applications, almost 94% of the applications are small and medium sized combined as shown in Figure 2. Only 6% of the applications have more than 50 components. Component distribution in the applications is shown in Figure 3. The highest number of applications, 1204 applications in the dataset, have 2 components each. Most of the applications (85%) have 1 to 30 components. Some applications have a large number of components. In the dataset, 17 applications have more than 200 components each. Two applications, WeChat (*com.tencent.mm*) and Alipay (*com.eg.android.AlipayGphone*), have more than 600 components each. We also found 26 applications without any components. These applications are mostly plugins, codecs, and libraries. The package names of some of these applications are *com.aviary.android.feather.plugins.borders.free*, *com.mxtech.ffmpeg.tegra3*, and *org.opencv.lib_v24_armv7a*. In Google Play Store, applications are identified by a package name, which we will use throughout the paper to identify an application. We will append the version name to the package name if available (*packageName-versionName* format). We have excluded 26 applications, which do not have any components, from further empirical studies unless explicitly specified.

4.2 Android applications' structure

Android applications are composed of activity, service, broadcast receiver, and content provider components. An activity represents a user screen or a user interface through which a user interacts with the application. Services are generally used to perform long-running background tasks, such as downloading files, and they do not provide a user interface. A broadcast receiver handles system or application generated events. To handle an event, a broadcast receiver needs to be registered for that event. Whenever the registered event triggers, the broadcast receiver is notified. For example, a broadcast receiver can be registered to be notified about system boot event. A content provider manages access to a structured set of data. In addition to these four types of components, an activity alias can be created in Android applications. In this paper, we treat an activity alias as a separate activity component because the attributes of an activity alias can be different from the attributes of its originating activity.

Applications in the dataset have a total of 241,052 components. Out of all the component types, the quantity of the activity component is significantly large (77%) as shown in Figure 4(a). Service, broadcast receiver, and content provider components combined constitute less than one-fourth of all the components. Only 421 components out of all the activity components are activity aliases. All activity aliases found in the dataset are restricted to 201 number of applications. The distribution of the activity aliases in those applications is shown in Figure 4(b). Out of the 201 applications, 133 applications have an activity alias each, whereas the highest number of activity aliases found in a single application *com.google.android.apps.plus-5.8.0.96635860* is 14. Excluding the 26 applications that do not have any components, only 41 applications out of 13,918 applications do not have an activity component as shown in Figure 4(c). Most of the applications that do not have an activity component belong to plugins or add-on applications, such

as *anonymous.plugin.luann-1.2*, *com.anysoftkeyboard.languagepack.alt.english-20100926*, and *com.appventive.tasksaddon-1.11*. Content provider components are the least used components in the applications. Only around 17% of the applications have at least one content provider component, while 33% of the applications in the dataset are composed of only activity components as shown in Figure 4(d). However, there are very few applications that contain only one type of components among service, broadcast receiver, and content provider. These applications belong to the same group of 41 applications where we could not find an activity component.



(a) Types of components in the dataset

(b) Activity alias

(c) No. of applications with a component type

(d) Applications with only one type of components

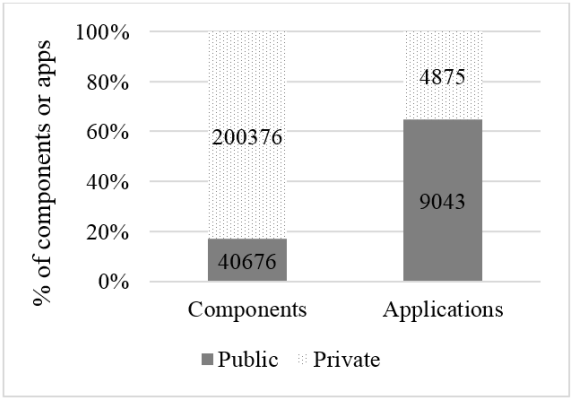
Figure 4. Applications composition in the dataset.

4.3 Public components

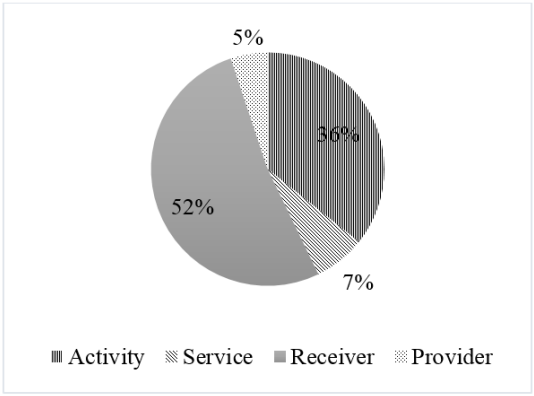
An Android application can expose its components for third-party applications. We define exposed components as public components and applications that have at least one public component as public applications. As discussed in Section 2, a component is declared public by setting its exported flag to true. In the case of activity, service, and broadcast receiver components, the default value of the exported flag depends on the intent filter. The presence of one or more intent filters set the flag to true. A content provider component is by default exported if the application uses an API level less than 17 by setting the value of *android:minSdkVersion* or *android:targetSdkVersion* attributes. An Android application has one main activity component, which starts when the application is launched. The main activity component is specified through an intent filter and is not considered as vulnerable. We exclude the main activity from the list of public components unless the main activity component also declares another intent filter.

As shown in Figure 5(a), only around 17% components in the dataset are public, but these public components are distributed in a large number of applications. Around 65% of the applications have at least one public component. Activity and broadcast receiver components constitute the major share of public components as shown in Figure 5(b). Although the broadcast receiver components constitute only 12% of the total components, it is the top contributor of

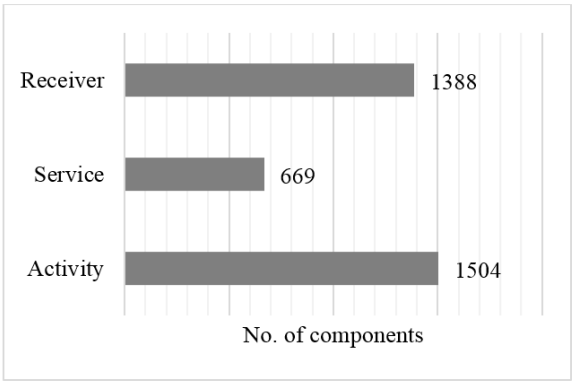
public components with 52%. Among the 14,529 public activity components, we found 363 activity aliases. Only 53 activity aliases are private in the dataset.



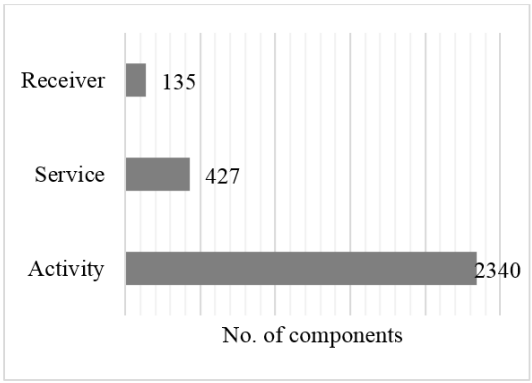
(a) Public components and applications



(b) Public component types



(c) Private components with intent filters



(d) Public components without intent filters

Figure 5. Public components and applications.

A component can be explicitly made private by setting its exported flag to false. The private components do not need to declare intent filters because they cannot receive explicit or implicit intents from another application. However, a large number of private components in the dataset declare intent filters. For example, some private components in *com.facebook.pages.app*, *com.google.android.youtube-10.21.58*, and *com.android.chrome-43.0.2357.93* applications declare intent filters. The distribution of these private components containing intents filters according to their type is shown in Figure 5(c). We found some use cases in the applications where it is appropriate. Components can receive implicit intents from the system. These components must declare intent filters. However, the system-originated implicit intents can also be received by private components. Therefore, it is not only appropriate to make these components private explicitly, but it also avoids malicious broadcast injection vulnerability. For example, if the components are public and they do not validate the received intents, then any application can send malicious explicit intents to those components. We also found some use cases where it is inappropriate. Private components are using non-standard actions or custom actions in the intent filter, which means that these components can only receive implicit intents sent by the same application. Although it does not have any known security vulnerabilities, it can create a reliability issue. Even if a component is private, intent filters declared by that component make it appear in the Android chooser list. If an application sends an implicit intent via chooser, which matches with the intent filter of the private component, then the chooser will display the name of the application containing the private component as one of the service providers. If a user selects the application, then the application will crash, throwing a security exception. Although this behavior has been corrected since Android 4.2, earlier versions still show the same behavior.

We also found a large number of components that explicitly set their exported flag to true, but do not declare any intent filters. For example, some public components in *com.amazon.kindle-4.13.0.185*, *com.avast.android.mobilesecurity-4.0.7886*, and *com.android.calendar-5.2.1-94626333* applications do not have an intent filter. The distribution of these public components according to their type is shown in Figure 5(d). This is an incorrect technique to declare public components, which make the components vulnerable. If a public component does not declare an intent filter, then it can be accessed only through explicit intents. Unlike private components, which can be accessed only through explicit intents generated by the same application, the public components without intent filters can be accessed through explicit intents generated by any application including malicious applications. The only restriction is that the applications must have prior knowledge of the component's class name. There is a large number of reverse engineering tools available for Android applications through which component's class name can be easily extracted. The vulnerability can result into both security and reliability issues. If the component is not protected by a permission, then it may cause a security issue. Meanwhile, if the component does not validate the received explicit intent, then it may cause reliability issue. The vulnerability can be avoided by either making these components private or protecting the components with a permission.

4.4 Permission-based protection

An Android application can communicate with both system applications, such as contacts and user applications. System applications are also referred as platform applications or stock applications. We refer user applications to those applications that are not system applications. Sensitive system applications and system resources are protected with system-defined permissions. For example, an application must have *READ_CONTACTS* permission to read contact information. Similarly, an application must have *INTERNET* permission to use the internet resource. However, a user application needs to define a custom permission to protect its exposed components. A custom permission is declared through a *permission* tag in the manifest file.

The applications in the dataset declare a total of 4,664 custom permissions as shown in Figure 6(a). Some applications declare the same permission more than once. We found 13 these applications in the dataset. The *com.animoca.google.astroboydash-1.4.3* application has two duplicate permissions whereas twelve other applications, such as *com.apcurium.MK.FlashCab-2.0.13* and *com.mobage.ww.a1903.SWTD_Android-1.3.1* have one duplicate permission each. The dataset also contains 437 duplicate custom permissions across applications, which mean that a permission declared by an application has also been declared by other applications. For example, *com.foursquare.permission.LOGIN* permission is declared by both *com.foursquare.robin-2015.06.08* and *com.joelapenna.foursquared-2015.05.28* applications. The duplicate permissions across applications also include those 14 permissions that were found duplicate within the same application. While declaring custom permissions, a common practice is to follow "*packageName.permissionName"* naming convention. However, the empirical results suggest that some developers are not following the practice, which results into duplicate custom permissions across applications. If two applications with a duplicate custom permission are installed in a device, the application that is installed first holds the permission, which means that if the second application has protected its component with the duplicate permission, then the first application can access the protected component without acquiring the permission of the second application. However, Android 5.0 and higher has mitigated this problem to some extent by allowing duplicate custom permissions to be declared only by applications signed with the same certificate. If the applications are not signed with the same certificate, then the installation of the second application will fail with an *INSTALL_FAILED_DUPLICATE_PERMISSION* error.

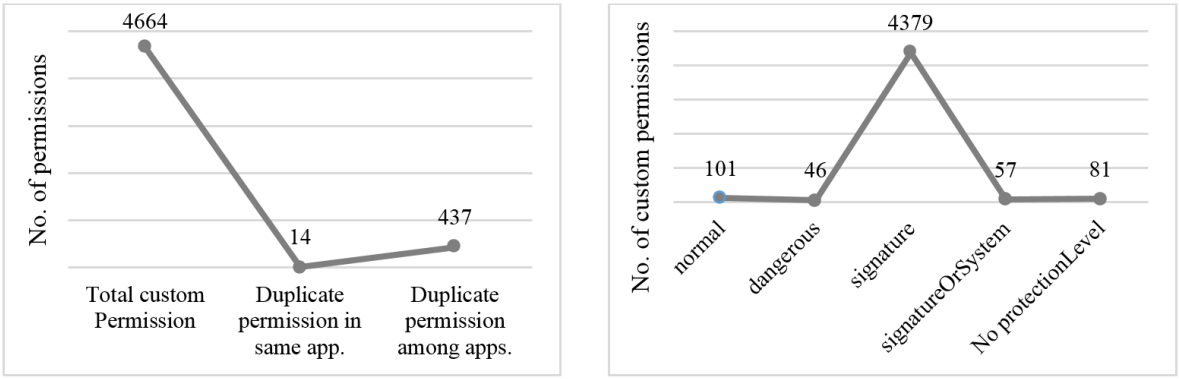
A large number of applications in the dataset declare a custom permission "<*packageName*>*.permission.C2D_MESSAGE*" where *packageName* indicates the package name of the declaring application. This custom permission is declared to avail Android Cloud to Device Messaging (C2DM) or Google Cloud Messaging (GCM) service. We found 2,820 permissions, which denote that around 20% applications in the dataset use C2DM or GCM service. It is important to note here that C2DM has been completely shut down since October 20, 2015 and only GCM service is allowed presently.

System permissions do not need to be declared by an application. However, some applications in the dataset, such as *com.google.android.googlequicksearchbox-4.7.13.16*, *com.frostbank.android-1.0.1*, and *com.weirdvoice-74* declare system permissions. We checked the permission declaration against 124 system permissions [19] and found

68 instances of system permission declaration from a group of 12 system permissions. The top three system permissions that have been declared similar to custom permissions are *FLASHLIGHT*, *INSTALL_SHORTCUT*, and *UNINSTALL_SHORTCUT*.

Android associates a protection level with a permission. The protection level signifies the level of security risk associated with the permission. There are four levels of protection: normal, dangerous, signature, and signatureOrSystem. A normal protection level is a minimum security risk permission that a user does not have to explicitly grant to an application because it is granted automatically by the system. A dangerous protection level is a higher security risk permission that requires explicit user consent. A permission with signature protection level can be used by only those applications that are signed with the same certificate. The signature-level permissions are silently granted by the system if the certificate matches. A permission with signatureOrSystem protection level is similar to a permission with signature protection level except that it can also be granted to those applications that reside in the system image.

The highest number of custom permissions in the dataset has signature protection level as shown in Figure 6(b). Only 46 custom permissions have dangerous protection level. Applications in the dataset also have 81 custom permissions without any protection levels. The undefined protection level belongs to normal protection level by default.



(a) Custom permissions in applications (b) Protection levels

Figure 6. Custom permissions.

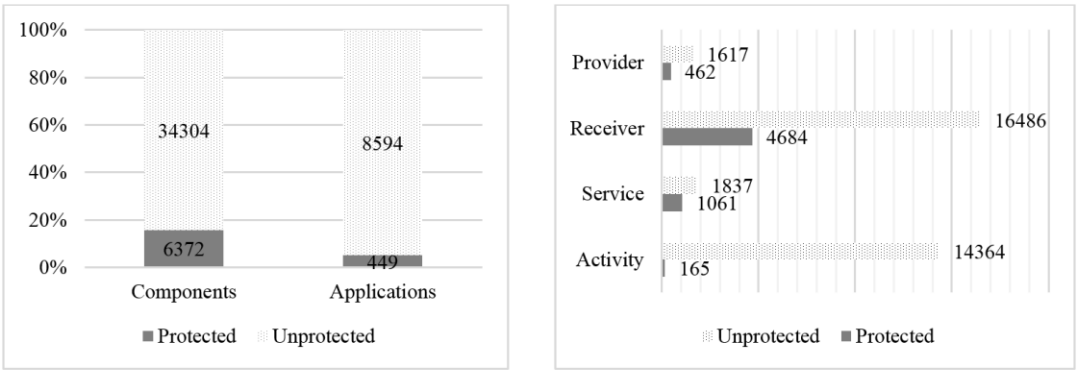
4.5 Protected public components

Public components of an Android application can be protected with permissions in the manifest file. The permission can be a system permission or a custom permission. For example, if a public component uses internet resource, which is a system-protected resource, then it is a good practice to protect the component with *INTERNET* permission to prevent unprivileged third-party applications from accessing privileged public components. Similarly, if a public component performs user-defined sensitive tasks, then the component needs to be protected with a custom permission. A permission is enforced at the component level using an *android:permission* attribute. In the case of content provider components, separate read and write permissions can be enforced at the component level using *android:readPermission* and *android:writePermission* attributes, respectively. The separate read and write permissions take precedence over the permission enforced through the *android:permission* attribute. We consider a content provider component protected if the permission is enforced by using any one of the aforementioned attributes. A content provider uses a content URI to identify its data. A permission in a content provider component can be enforced at the path level to protect a content URI. If a public content provider component is not protected at the component level but only protected at the path level, then we treat this content provider component as unprotected. Components can also be protected at the application level by enforcing a permission through an *android:permission* attribute in the <*application*> element of the manifest file. The application-level permission protects all the public components of the application. However, the component level permission overrides the application-level permission.

As shown in Figure 7(a), around 16% of public components are protected with permissions that leave a large number of components unprotected in the dataset. Only five applications in the dataset, such as *bto.apps.drugdealer-*

2.26, *com.Teartek.MatrixWallpaper-3.4*, and *com.appventive.mailprovider-0.4*, are protected at the application level. At the component level, we found both system and custom permission protected components. Out of 6,372 protected components, 1,254 components are protected with the system permissions whereas only 854 components are protected with the custom permissions. This leaves a large number of protected components that neither use the system permissions nor use the declared custom permissions. After investigation, we found 3,968 broadcast receiver components protected with the *com.google.android.c2dm.permission.SEND* permission. The messages received from GCM are handled by a broadcast receiver component that must be protected with the *com.google.android.c2dm.permission.SEND* permission. The permission "<*packageName*>*.permission.C2D_MESSAGE*" is declared only to prevent other applications from registering and receiving the messages. Out of 296 remaining protected components, a large number of broadcast receiver components are protected with the *com.amazon.device.messaging.permission.SEND* permission, which is used for Amazon Device Messaging (ADM) Service. Many components are protected with the system permissions that are not listed in [19], such as *MANAGE_USERS* and *BIND_JOB_SERVICE*. Some components are protected with custom permissions that are possibly declared in other applications and are not available in the dataset.

We found some applications in which the *android:permission* attribute is used incorrectly, which leaves the applications unprotected and at security risk. In four cases, such as *com.teaandtoys.unshorten-0.2* and *com.tencent.mm-6.2.0.53_r1166628*, the value of the *android:permission* attribute is empty. In another two cases, *com.ncaa.mmlive.app-4.0.1* and *com.tour.pgatour-4.3*, the value of the *android:permission* attribute is assigned as *false*. We also found 25 applications, such as *airborne.nbawp-3.1*, *com.discoverfinancial.mobile-6.3.1*, and *com.ibm.events.android.usga-3.0* in which the *android:permission* attribute is declared in the <*intent-filter*> elements. After manual observation, we found that all these 25 applications use IBM Push Notification service. The documentation [33] for using the service incorrectly states that the *android:permission* attribute should be used in the <*intent-filter*> element rather than the <*receiver*> element.



(a) Protected vs unprotected components and apps (b) Protected vs unprotected component types

Figure 7. Protected public components and applications.

In the dataset, only 2,820 applications declare the "<*packageName*>*.permission.C2D_MESSAGE*" permission, but 3,968 public components are protected with the *com.google.android.c2dm.permission.SEND* permission. After investigation, we found that some applications, such as *air.com.classteacher.main-7.35* and *com.activision.callofduty.heroes-1.7.1*, are protecting more than one broadcast receiver components with the permission, which leaves 3,746 unique applications protecting their broadcast receiver components with the *com.google.android.c2dm.permission.SEND* permission. Nevertheless, there are 926 applications that do not declare the "<*packageName*>*.permission.C2D_MESSAGE*" permission, but their public components are protected with the *com.google.android.c2dm.permission.SEND* permission. Further investigation shows that 242 applications, such as *airborne.nbawp-3.1* and *com.bianor.amspersonal-1.0.5* declare GCM permission with <*packageName*> different from the package name defined in the *manifest* tag. The remaining 684 applications, such as *com.gmail.playmood.puzzletoys-1.1*, *com.funny.jokes.image.gallery-1.9*, and *vasilediu.stopcrows.free-1.9* do not declare the GCM permission, but protect their public components with the *com.google.android.c2dm.permission.SEND* permission.

We also found 1,151 private components protected with permissions. The private components include those components that set the exported flag to false explicitly and do not have an intent filter, set the exported flag to false explicitly and have intent filters, and neither set the exported flag explicitly nor have any intent filters. For example, the *com.cp.mpos* application protects all of its private components with permissions. Some of the components in the applications, such as *com.dollarbank.onlinebanking-2.4.2*, *com.facebook.groups*, and *com.yahoo.mobile.client.android.im-1.8.8* have intent filters with exported flag set to false. However, those private components are protected with permissions.

Some system-generated intents are protected with the system permissions. An application can receive these intents by declaring the appropriate intent filters and the corresponding permissions. A large number of applications in the dataset declare intent filters to receive the protected intents. However, the applications have not declared the required permissions. For example, 681 applications, such as *com.facebook.groups*, *com.google.android.apps.authenticator2-2.49*, and *yong.universalplayer-3.2.3*, declare the intent filter to receive *android.intent.action.BOOT_COMPLETED* intent, but they do not use the *android.permission.RECEIVE_BOOT_COMPLETED* permission. Although it does not create any serious security and reliability issues, the applications will not be able to receive the protected intents. Hence, the applications will not perform the intended task.

Only 449 public applications in the dataset are completely protected, which means that all of their public components are protected with permissions. Among the 8,594 unprotected applications, 4,350 applications have none of their public components protected. The remaining 4,233 applications are partially protected. The distribution of the protected components according to their type is shown in Figure 7(b).

4.6 Sensitive public applications

The definition of public applications from Section 4.3 is further refined as those applications that have at least one unprotected public component. An application is considered as a sensitive application if it performs a task that directly or indirectly involves sensitive user data and may cause personal or monetary loss. For example, reading contact information involves sensitive user data, such as email addresses and phone numbers, which may cause personal loss if stolen. Similarly, sending SMS may cause monetary loss if the recipient number is a premium rate number. Collecting user data and sending premium-rate SMS messages are the most common malicious activities [10]. Sensitive system and application resources are protected with system permissions and custom permissions, respectively. The protection level of a permission further defines the level of security risk associated with the protected sensitive resources. Out of four protection levels discussed in Section 4.4, only permissions with the dangerous protection level is considered as high-security risk permissions. We define a sensitive public application as an application that has at least one unprotected public component and uses at least one dangerous system or custom permission.

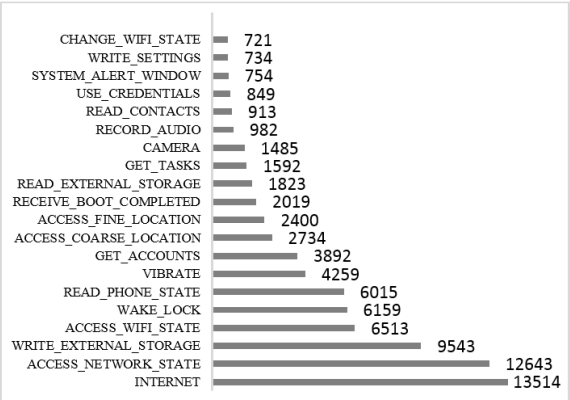


Figure 8. Frequently used system permissions.

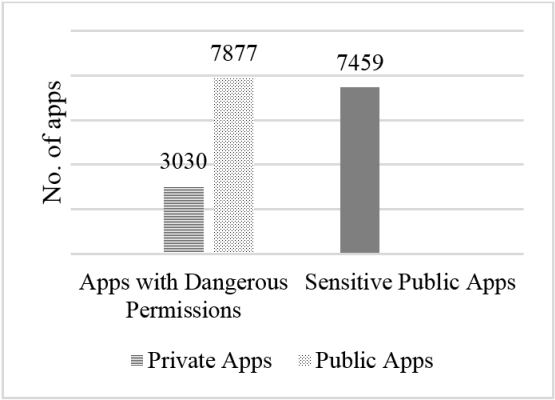


Figure 9. Sensitive applications.

An Android application uses permission through the <*uses-permission*> element in the manifest file. Only 400 applications in the dataset do not use any permissions. System permissions at [19] are not comprehensive. To obtain the complete list of system permissions, we extracted all the unique occurrences of system permissions used in the applications of the dataset. The system permissions can be identified by its namespace as *android.permission.** except in few cases. We extracted a total of 321 unique system permissions. After manually checking all these permissions, 163 incorrect permissions were found. Most of the incorrect permissions appear to originate from developer mistakes such as, *android.permission.BIND.WALLPAPER* instead of *android.permission.BIND_WALLPAPER*, *android.permission.ACCESS_FINE_LOCATIION* instead of *android.permission.ACCESS_FINE_LOCATION*, and *android.permission.GET_TASK* instead of *android.permission.GET_TASKS* in *app.cobo.launcher-1.3.9.5*, *com.wine.chroisen2eng-1.0.5*, and *com.CoolTopFreeGamesandApps.moneytime-1.1* applications, respectively. We also found 41 applications, such as *com.allso.risewarsads-7.4*, *com.nicegame.racing-1.06*, and *com.polarbit.rthunderlite-1.2.0* in which the <*uses-permission*> element is declared inside the <*application*> element whereas it should have been declared outside the <*application*> element in the manifest file. If an application tries to access resources protected by the incorrectly used system permission, then the application will crash by throwing a security exception. After removing the incorrect permissions, we obtained a total of 158 unique system permissions. We then derived the final list of 178 unique system permissions by combining the permissions at [19] and the permissions extracted from the applications. This permission list is still not exhaustive, but contains all the system permissions used by the applications in the dataset. We performed a short analysis on frequently used system permissions. The top twenty frequently used system permissions in the dataset are shown in Figure 8.

Among the 178 unique system permissions, 23 system permissions are dangerous permissions. Owing to the common use of some system permissions, such as *INTERNET* and *CHANGE_WIFI_STATE*, their protection level has been downgraded from dangerous to normal. Some system permissions, such as *USE_CREDENTIALS*, have been removed but temporarily kept with downgraded protection level for backward compatibility. Obtaining a comprehensive list of dangerous custom permissions is a non-trivial task. As of July 2017, Google Play Store has more than 3 million applications [20] and each of those applications can declare dangerous custom permissions. In this paper, we limit the discussion to dangerous custom permissions declared by only those applications that are present in the dataset. As discussed in Section 4.4, there are 46 dangerous custom permissions declared by the applications in the dataset.

A sensitive application in the dataset is an application that uses at least one of the 69 (23 system and 46 custom) dangerous permissions. We found a total of 10,907 sensitive applications, where 3,030 applications are private as shown in Figure 9. Private applications do not expose any components. Out of the 7,877 public applications that use one or more dangerous permissions, 418 applications are completely protected, which leaves 7,459 applications in the dataset as sensitive public applications.

5. Discussion

In Section 4, we performed several empirical studies and obtained various results. In this section, we interpret these results in the context of collaborative model and its security risk.

5.1 Scale of collaboration

Collaboration in Android applications is achieved through public components. The empirical results in Section 4.3 suggest that a large number of applications (65%) participate in collaborative work through comparatively less number of components (17%). It indicates that a large number of applications are mostly performing their tasks through private components while still offering some services to third-party applications. If we take the component types into account, then the broadcast receiver components are clearly on the top of the list of public components with 52%. Although the broadcast receiver components participate in the collaborative work, they do not offer any services to third-party applications. They receive a system or an application event, and then either perform a very short task themselves or start other components to perform a task. The tasks, either performed by themselves or through other components, are generally private to an application. If we limit the collaborative model in terms of services offered by applications, then only 45% of the applications participate in this limited version of the collaborative model through

8% components. Activity components are the most frequently used components for offering services to third-party applications.

Although the definition of public components, specifically for activity components, stated in Section 4.3 is absolutely correct, all public activity components cannot collaborate openly. When an implicit intent is used to start an activity component, and if the intent has not been placed into any categories, then the system places it into *CATEGORY_DEFAULT*. This means that a public activity component of an application willing to receive an implicit intent from another application must declare at least "*android.intent.category.DEFAULT*" as a category in its intent filter. The public activity components that do not declare at least one category in their intent filter can only be accessed through explicit intents. Given that an explicit intent requires prior knowledge of the target component name, it imposes certain restrictions in the collaboration. We found 4,112 activity components that are public but do not declare a category. If we further limit the collaborative model by not considering the collaboration through explicit intents, then around 41% of the applications participate in the collaborative model through only 6% components. Nevertheless, activity components take the lead role in the collaboration.

Android applications are primarily designed for a very specific purpose. However, 41% (27% small, 61% medium, and 12% large) of the applications in the dataset are offering their services to third-party applications through 6% (68% activity, 19% service, and 13% content provider) components. Although it is not the subject of investigation in this paper, the major motivations behind offering services could be monetary gain through advertisements and popularity gain for the application. Regardless of the reasons, offering services helps both developers and user communities of Android. While developers can rapidly develop applications by taking advantage of the existing collaborative applications, users get the option to select the best application among the collaborative applications to avail services. In summary, we can state that the collaborative model has been implemented in a large number of applications and is contributing to the success of Android.

5.2 Security risk assessment

In Section 5.1, we restricted the definition of the collaborative model in terms of services offered by an application to third-party applications. In this section, we fall back to the original definition in which all the public components participate in some form of collaboration. By this definition, 65% of the applications collaborate through 17% public components, which denote that 65% of the applications in the dataset are at security risk if all of their public components are not protected with permissions. The empirical results in Section 4.5 show that only around 5% of the applications among all the public applications are completely protected, which leaves a large number of applications (62%) unprotected and at security risk.

Although 62% of the applications are at security risk, compromise on these applications may not cause severe damage unless the applications are performing sensitive tasks. The empirical results in Section 4.6 show that around 57% of public applications in the dataset are performing sensitive tasks. Among these applications, only 418 applications are completely protected, which means that 54% of the applications are at serious security risk and compromise on these applications may cause financial or personal loss to users. As discussed in Section 2, all public applications (62%) can become accessories in privilege escalation attack if they have the privilege of accessing protected components. Thus, even protected public applications can be at security risk indirectly.

The level of security risk can be further assessed depending on the type of components used for collaboration. An activity component is the most preferred component type for collaboration because it presents a UI screen and the task is performed through user interactions. An observant user can detect malicious behavior and minimizes the security risk. Meanwhile, collaboration through a service component maximizes the security risk because service components are executed in the background without user attention. The empirical results in Section 4.3 show that collaboration is mostly performed through activity (36%) and broadcast receiver (52%) components, which are positive indications. However, the empirical results in Section 4.5 show that most of the activity (99%) and broadcast receiver (78%) components are not protected. Broadcast receiver components are mostly used to receive system events. Although an application cannot generate system events, an unprotected public broadcast receiver supposed to receive only system events can be highly vulnerable to malicious broadcast injection [11]. Service (7%) and content provider (5%) components are the least used components for collaboration, which are again positive indications. Compared to other component types, a large number of services (37%) are protected, which means that developers are aware of the

security risk associated with the collaboration through service components. However, 63% of the services are unprotected, which is extremely dangerous for the applications in terms of security.

5.3 Security risk mitigation

The collaborative model in Android always invites direct or indirect security risk. As discussed in Section 2, even securely implemented collaboration has an indirect security risk. The empirical results in this paper indicate that a large number of applications perform collaboration. Under these circumstances, the first step for developers is to avoid the security risk caused by unnecessary collaboration. For example, a large number of applications expose components while collaborating with the system, which is not required. Applications can collaborate with the system without exposing components. One best practice is to set the exported flag to false explicitly if the component is not supposed to receive intents from other applications. Our empirical results in Section 4.3 suggest that some developers are following this practice, which avoids the security risk.

Exposing components for collaboration is unavoidable under many circumstances. In these cases, developers should avoid the use of sensitive resources, particularly protected by dangerous permission. This makes the applications less vulnerable for attackers even if the components are unprotected because the attackers cannot access sensitive resources. However, our empirical results in Section 4.6 suggest that a large number of applications with one or more exposed components use sensitive resources. If developers cannot avoid exposing components and using sensitive resources, then developers should protect the exposed components. However, our empirical results suggest that very few applications that use sensitive resources are completely protected. Developers can use some existing tools, such as Android Lint [35] integrated with Android Studio and ComDroid [11], which provide warnings against unprotected exposed components.

Following the best practices for security and privacy [36] does not guarantee risk-free applications. Developers have to implement the security correctly in the applications. In the empirical study, we found several mistakes made by developers in implementing permission-based security for the collaborative model. Therefore, developers should ensure that security has been implemented correctly. In addition to tools, such as Android Lint [35] and ComDroid [11], which detect unprotected components and other security vulnerabilities, developers can use our tool ManifestInspector [37], which is an open source rule-based static analysis tool that detects all kinds of errors in the Android manifest file including errors in implementing the collaborative model and its security.

6. Threats to validity

The empirical studies have been performed on top popular applications from all categories present in the Google Play Store. Although the dataset represents all kinds of applications, it may not represent those applications that are comparatively low quality, which may affect the empirical results related to security risk. All empirical results are entirely based on the analysis of manifest files of the applications. Any kind of source code analysis has not been performed in this paper, which may result in some deficiency. A broadcast receiver component can be declared in the source code in addition to the manifest file. Security of these dynamic broadcast receiver components is also implemented in the source code. Our empirical studies oversight all aspects of dynamic broadcast receiver components. Android applications collaborate in two different ways: by receiving intents from other applications through exported components and by sending intents to other applications that have exported components. Given that we have not analyzed the source code, our empirical results do not include collaboration that is achieved through sending intents to other applications. However, one application must have an exported component to achieve any kind of collaboration between applications.

In Section 4.6, we defined a sensitive public application as an application that has at least one unprotected public component and uses at least one dangerous system or custom permission. Although we managed to identify all the dangerous system permissions used by the applications in the dataset, we could not obtain the comprehensive list of dangerous custom permissions, which may affect the empirical results of sensitive applications. Our definition of sensitive public application is broad. We assumed that there is always a path in an application from an unprotected

public component to resources protected by dangerous permissions. In reality, this path may not always exist. For accuracy, reachability analysis is required.

7. Conclusion

We performed several empirical studies focusing on the collaborative model and its security risk. The empirical results suggest that a large number of Android applications are small to medium-sized applications. Very small number of applications are large-sized. Irrespective of application size, a large number of applications (65%) participate in collaboration with maximum frequency among medium-sized applications. Although collaboration is performed by a large number of applications, comparatively fewer applications (41%) offer services to third-party applications. While collaboration is achieved mainly through broadcast receiver components, applications offer services mainly through activity components. In summary, the empirical results suggest that the collaborative model has succeeded in Android applications.

The study of the collaborative model in Android cannot be completed without the study of its security. The collaborative model always invites direct or indirect security risk and it is up to the developers to mitigate the risk. The empirical results suggest that a large number of applications (54%) are at serious security risk, which means that developers are either not aware of security issues associated with the collaborative model or implementing security incorrectly. In the empirical study, we observed that developers are making several mistakes in implementing the security. This study clearly indicates that the success of the collaborative model can be hindered by its high-security risk.

Acknowledgements

This research was supported by the BK21 Plus project (SW Human Resource Development Program for Supporting Smart Life – 21A20131600005) and the Basic Science Research Program through the National Research Foundation of Korea (No. NRF-2017R1D1A3B04035880) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea.

References

[1] Smartphone OS market share - http://www.idc.com/prodserv/smartphone-os-market-share.jsp [accessed on July 5, 2017].

[2] Revenue for developers in Google play store vs iOS app store - http://www.pcmag.com/article2/0,2817,2498161,00.asp [accessed on July 5, 2017].

[3] Kantola, David, Erika Chin, Warren He, and David Wagner. "Reducing attack surfaces for intra-application communication in android." In Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, pp. 69-80. ACM, 2012.

[4] Davi, Lucas, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. "Privilege escalation attacks on android." In Information Security, pp. 346-360. Springer Berlin Heidelberg, 2011.

[5] Felt, Adrienne Porter, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. "Permission Re-Delegation: Attacks and Defenses." In USENIX Security Symposium. 2011.

[6] Intents and Intent Filters - http://developer.android.com/guide/components/intents-filters.html [accessed on July 5, 2017].

[7] Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards Taming Privilege-Escalation Attacks on Android." In NDSS. 2012.

[8] Enck, William, Machigar Ongtang, and Patrick McDaniel. "Understanding android security." IEEE security & privacy 1 (2009): 50-57.

[9] Shabtai, Asaf, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. "Google android: A comprehensive security assessment." IEEE Security & Privacy 2 (2010): 35-44.

[10] Felt, Adrienne Porter, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. "A survey of mobile malware in the wild." In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, pp. 3-14. ACM, 2011.

[11] Chin, Erika, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing inter-application communication in Android." In Proceedings of the 9th international conference on Mobile systems, applications, and services, pp. 239-252. ACM, 2011.

[12] Enck, William, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. "A Study of Android Application Security." In USENIX security symposium, vol. 2, p. 2. 2011.

[13] Octeau, Damien, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis." In USENIX Security 2013. 2013.

[14] Bagheri, Hamid, Alireza Sadeghi, Joshua Garcia, and Sam Malek. "Covert: Compositional analysis of android inter-app permission leakage." IEEE transactions on Software Engineering 41, no. 9 (2015): 866-886.

[15] Lu, Long, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. "Chex: statically vetting android apps for component hijacking vulnerabilities." In Proceedings of the 2012 ACM conference on Computer and communications security, pp. 229-240. ACM, 2012.

[16] Li, Li, Alexandre Bartel, John Klein, and Yves Le Traon. "Automatically exploiting potential component leaks in android applications." In IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 388-397. IEEE, 2014.

[17] Tor Browser - https://www.torproject.org/projects/torbrowser.html.en [accessed on July 5, 2017].

[18] Apktool - https://ibotpeaches.github.io/Apktool/ [accessed on July 5, 2017].

[19] System Permissions - http://developer.android.com/reference/android/Manifest.permission.html [accessed on November 30, 2016].

[20] AppBrain - Number of available android applications in the Play Store. http://www.appbrain.com/stats/number-of-android-apps. [Accessed on July 5, 2017].

[21] Barrera, David, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. "A methodology for empirical analysis of permission-based security models and its application to android." In Proceedings of the 17th ACM conference on Computer and communications security, pp. 73-84. ACM, 2010.

[22] Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. "Android permissions demystified." In Proceedings of the 18th ACM conference on Computer and communications security, pp. 627-638. ACM, 2011.

[23] Au, Kathy Wain Yee, Yi Fan Zhou, Zhen Huang, and David Lie. "Pscout: analyzing the android permission specification." In Proceedings of the 2012 ACM conference on Computer and communications security, pp. 217-228. ACM, 2012.

[24] Bartel, Alexandre, John Klein, Martin Monperrus, and Yves Le Traon. "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android." IEEE Transactions on Software Engineering 40, no. 6 (2014): 617-632.

[25] Maji, Amiya K., Fahad Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. "An empirical study of the robustness of inter-component communication in Android." In 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1-12. IEEE, 2012.

[26] Sasnauskas, Raimondas, and John Regehr. "Intent fuzzer: crafting intents of death." In Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), pp. 1-5. ACM, 2014.

[27] Bouncer - http://googlemobile.blogspot.kr/2012/02/android-and-security.html [accessed on July 5, 2017].

[28] Oberheide, Jon, and Charlie Miller. Dissecting the android bouncer. In SummerCon, 2012.

[29] Chen, Kai, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale." In 24th USENIX Security Symposium, pp. 659-674. 2015.

[30] BrainTest – A New Level of Sophistication in Mobile Malware - http://blog.checkpoint.com/2015/09/21/braintest-a-new-level-of-sophistication-in-mobile-malware/ [accessed on July 5, 2017].

[31] Zhou, Yajin, Zhi Wang, Wu Zhou, and Xuxian Jiang. "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets." In NDSS. 2012.

[32] Verify apps - https://support.google.com/accounts/answer/2812853?hl=en [accessed on July 5, 2017].

[33] IBM Push Notification - http://developer.xtify.com/display/sdk/Getting+Started+with+Google+Cloud+Messaging [accessed on November 30, 2016].

[34] Jha, Ajay Kumar, and Woo Jin Lee. "Analysis of Permission-based Security in Android through Policy Expert, Developer, and End User Perspectives." Journal of Universal Computer Science 22, no. 4 (2016): 459-474.

[35] Android Lint - https://developer.android.com/studio/write/lint.html [accessed on July 5, 2017]

[36] Best Practices for Security & Privacy - https://developer.android.com/training/best-security.html [accessed on July 5, 2017]

[37] Jha, Ajay Kumar, Sunghee Lee, and Woo Jin Lee. "Developer mistakes in writing Android manifests: an empirical study of configuration errors." In Proceedings of the 14th International Conference on Mining Software Repositories, pp. 25-36. IEEE Press, 2017.

Vitae

Ajay Kumar Jha is currently a Postdoctoral Researcher at Kyungpook National University. He received his MS and PhD in Computer Science from Kyungpook National University, South Korea in 2013 and 2017, respectively. His main research interests include software engineering and mobile security. He is specifically interested in software testing, program analysis, empirical software engineering, and security of Android applications.

Woo Jin Lee is currently a professor in the school of Computer Science and Engineering at Kyungpook National University, South Korea. He received his PhD degree in Computer Science from Korea Advanced Institute of Science and Technology in 1999. His main research interests include software testing, requirements engineering, and embedded systems.