revDroid: Code Analysis of the Side Effects after Dynamic Permission Revocation of Android Apps

Zheran Fang, Weili Han, Dong Li, Zeqing Guo, Danhao Guo, Xiaoyang Sean Wang
Fudan University
wlhan@fudan.edu.cn

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Hao Chen
ShanghaiTech University
chenhao@shanghaitech.edu.cn

ABSTRACT
Dynamic revocation of permissions of installed Android applications has been gaining popularity, because of the increasing concern of security and privacy in the Android platform. However, applications often crash or misbehave when their permissions are revoked, rendering applications completely unusable. Even though Google has officially introduced the new permission mechanism in Android 6.0 to explicitly support dynamic permission revocation, the issue still exists. In this paper, we conduct an empirical study to understand the latest application practice post Android 6.0. Specifically, we design a practical tool, referred to as revDroid, to help us to empirically analyze how often the undesirable side effects, especially application crash, can occur in off-the-shelf Android applications. From the analysis of 248 popular applications from Google Play Store, revDroid finds out that 70% applications and 46% permission-relevant calls do not appropriately catch exceptions caused by permission revocation, while third-party libraries pay much more attention to permission revocation. We also use revDroid to analyze 132 recent malware samples. The result shows that only 27% malwares and 36% permission-relevant API calls of malwares fail to consider the permission revocation. In fact, many of them perform specialized handling of permission revocation to keep the core malicious logic running. Finally, revDroid can be used to help developers uncover the unhandled permission revocations during development time and greatly improve the application quality.

Keywords
Android Security, Permission Over-claim, Permission Revocation, revDroid

1. INTRODUCTION
The security of Android devices heavily depends on the effectiveness of the permission mechanism [5, 19, 20]. The permission mechanism of Android is often considered a severe design flaw of the main-stream Android platform on the current market, because users usually have to approve all the permission requests during the application installation process. Specifically, the over-claim of permission is a common issue in the ecosystem of Android applications. Many reasons, including poor documentations [21], coarse-grained permissions [9], even irresponsible developers [9], all contribute to this issue. Unfortunately, users are basically given the all-or-nothing model.

A practical solution is to revoke permissions separately after an application is installed. Android 4.3 introduces an experimental system service called AppOps [14], which can help users to revoke a subset of permissions after applications are installed. This functionality of permission revocation has been provided officially in Android 6.0 [4]. For applications targeted at Android 6.0, an Android user does not need to grant any permissions at install-time. The application will request the permissions when it actually needs them at run-time by showing a dialog to ask for the permission. Users can also grant or revoke permissions after installation. Besides these, researchers proposed many frameworks and mechanisms in the framework layer or the kernel layer by allowing users to selectively grant permissions to an application, e.g., Android Permission Extension (Apex) [31], MockDroid [11] and AppFence [28].

However, after the permission revocation, applications would typically crash if they are not written to support permission revocation, resulting in undesirable user experience. This issue could be more serious when the new platform of Android 6.0 explicitly enables the dynamic permission revocation. Based on log analysis of application crash, the tool Pyandrazzi [29] reports that around 5.8% applications crash after a permission is removed (which we believe is a significant underestimate).

There are a few existing and practical methods to mitigate the side effects. Firstly, researchers attempt to forge the data [11, 28, 39, 38] that the applications request. Secondly, system status, e.g., enabling airplane mode, can be forged when the applications access corresponding data or perform corresponding operations [28]. Some of these optimization methods are adopted in Android 6.0, but we find that they only eliminate application crashes for certain APIs.

We are curious about the real possibilities of application crashes under Android 6.0. We find that the real situation is worse than previously believed, because Pyandrazzi only
analyzed the log reported by logcat [3] when running the applications dynamically, which suffers from coverage issues, because many program paths which lead to the crash are not run.

In this paper, we show that static code analysis can provide a much more accurate estimation.

The main contributions of this paper are as follows:

- To analyze the side effects after dynamic permission revocation of Android applications, we design a practical tool, referred to as revDroid, based on Soot [34] and FlowDroid [7]. revDroid can automatically count the reachable but unhandled SecurityException in an Android application as the cause of side effects of permission revocation. revDroid counts an unhandled permission-relevant invocation, if the invocation does not belong to either one of the following types: i) the invocation itself or ancestral call methods are wrapped by a try and catch block which handles the SecurityException; or ii) the invocation itself or ancestral call methods are wrapped by a proactive permission check block which calls permission check methods such as checkPermission and checkSelfPermission.

- After using revDroid to successfully analyze 248 top popular applications from Google Play Store, we are surprised to find that 70% applications and 46% permission-relevant API calls fail to consider permission revocation. We did experiments to confirm that the uncaught SecurityException is bound to cause crash. When we analyze the unhandled SecurityExceptions in four categories of third-party libraries of Android application, we find that third-party libraries pay more attention to correctly invoke permission relevant APIs. These results show that the problem is much more severe than previously concern [29]. This high rate of unhandled SecurityException also implies that there is still much work needed from application developers to meet the new requirement and a tool such as revDroid can help significantly to ease the transition.

- We also use revDroid to analyze 132 recent malware samples from the M0Droid project [18] by Damshenas et al. The result shows that 27% malwares and 36% permission-relevant API calls fail to consider the permission revocation, which is surprisingly better compared to top regular applications. After careful inspection, we believe that malware samples in fact do deal with permission revocation in order to keep as much malicious logic running as possible.

In summary, we believe the tool and analysis results are valuable after Android 6.0 is released, as they can tell how serious this issue of dynamic permission revocation is when the developers migrate the state-of-the-art applications to the new platform. The tool can also help developers and markets identify uncorrecting applications.

The rest of this paper is organized as follows: Section 2 introduces the background knowledge and motivation of this paper. Section 3 designs our code analysis framework. Section 4 presents our analysis experiments and results. Section 5 discusses the remaining issues. Section 6 introduces the related work. Section 7 summarizes the paper and introduces our future work.

2. BACKGROUND AND MOTIVATION

2.1 Over-claim of Permissions and Its Countermeasures

Over-claim of permissions is very popular in the current Android platform [19, 33], because developers and application users might have conflict of interest [27]. Developers tend to declare more permissions than necessary because more permissions will make the development process easier, allow applications to access more private information and perform more critical operations. However, application users who tend to protect their privacy may not know what the requested permissions mean [23]. Further, before Android 6.0, users have no other choice but to approve all the permission requests. The issue is made even more severe by the coarse-grained permissions and insufficient documentations [20].

The over-claim of permissions breaks the principle of least privilege (PLP) [32]. This violation of PLP exposes users to potential privacy leakage and financial losses directly or indirectly. For example, if a standalone game application requests the SEND_SMS permission which could be unnecessary, the permission can be exploited to send premium rate messages without users’ acknowledgment. As discovered by Felt et al., about one-third of 940 applications analyzed were over-privileged, and the most common unnecessary permissions include ACCESS_NETWORK_STATE, READ_PHONE_STATE, ACCESS_WIFI_STATE, WRITE_EXTERNAL_STORAGE, and CALL_PHONE [21]. Au et al. also identified that 543 out of 1,260 applications required at least one over-claimed permission [8]. In the new permission mechanism of Android 6.0, 28 permissions falling under PROTECTION_DANGEROUS are divided into 9 permission groups according to their functionalities, and permissions in the same permission group are managed as a whole, which makes the permission granularity much coarser [4]. This would lead to the issue of the over-claim of permissions as well.

A major category of countermeasures to the issue is to allow users to revoke over-claimed permissions at install-time or run-time. Researchers have proposed several ways to achieve this goal. For example, Android Permission Extension (Apex) [31] and Flex-P [30] augmented the Android application installer to allow users to selectively grant or revoke permissions, instead of granting or revoking permissions all together. In addition, a permission editor was also provided to allow users to grant more permissions or revoke some of the granted permissions even after the application has been installed. This approach required heavy modification to the Android operating system itself, but yielded a flexible solution [29]. In Android 6.0, Google introduced a new permission mechanism, where users can deny permission requests at run-time and still continue to use the application [25].

2.2 Side Effects of Permission Revocation and Fake Data Methods

Before Android 6.0, Google suggests that the dynamic permission mechanism would be too much of a burden on the user, so the Android documentation does not explicitly instruct application developers to handle cases of permission revocation [29], and even suggests that application developers not to worry about run-time failures caused by missing permissions [5]. In almost all cases the revocation of permis-
sions will result in a Java SecurityException being thrown back to the application, although it is not guaranteed everywhere (for example, the sendBroadcast(Intent) method checks permissions after the method call has returned and no exception will be thrown even if there are permission failures) [5]. Although all the application developers will handle the exceptions gracefully, most developers typically develop applications according to the stock Android permission mechanism before Android 6.0 and assume that all the permissions which his or her application requests are granted when the application is running on users’ devices. If the SecurityException is not handled, applications are likely to malfunction, resulting in undesirable user experience, such as UI freezing, data corruption, or even complete crash.

As evaluated by Kennedy et al., 39 (5.9%) of the 662 applications crashed due to permission revocation overall but not all permissions were equal. Removing the READ_CONTACTS and ACCESS_FINE_LOCATION permissions had the greatest impact which caused 20 and 13 applications to crash respectively, while removing the CAMERA, RECORD_AUDIO and WRITE_SMS permissions respectively never caused crash [29].

To prevent SecurityException from being thrown, many previous studies, such as MockDroid [11], AppFence [28] and TISSA [39], leveraged the idea of fake data. That is, when the user revokes a permission, the Android operating system returns fake data to the application, instead of simply denying the access. For example, the unique identifiers of an Android device (IMEI or IMSI) can be substituted with fake ones, and the geographic coordinate of a fixed place can replace that of the device’s real location. This method attempted to protect the user’s privacy while maintaining usability.

After Android 6.0 is released, although Google changes the Android documentation to explicitly instruct application developers to handle cases of permission revocation, it will take application developers a lot of time and effort to manually examine and modify their applications to support the new permission mechanism. Moreover, even in the new permission mechanism, if the user revokes a permission and the application tries to use a functionality that requires that permission, the Android operating system will also throw a SecurityException to the application for some APIs, for example, the getDeviceId() method of the android.telephony.TelephonyManager class which is protected by the READ_PHONE_STATE permission.

2.3 Soot and FlowDroid

RevDroid is designed and implemented based on Soot [34] and FlowDroid [7]. Soot is an analysis and transformation framework developed by McGill University, which supports input formats including Java byte-code, Java source code and Android byte-code. Soot can also produce transformed code in output formats such as Java byte-code and Android byte-code. Soot provides powerful analysis functionalities including call graph construction, dead code elimination, point-to-analysis, def/use chain analysis and data flow dominator analysis. FlowDroid is a context-sensitive, flow-sensitive, object sensitive and lifecycle-aware static analysis for Android applications based on Soot. Compared with Soot, FlowDroid provides functionalities specific to Android applications, including resource file parsing, UI mapping, callback calculation and entry point generation based on the application components. FlowDroid aims for an analysis with very high recall and precision [7].

2.4 Application Scenarios

Application developers and markets, e.g., Google, may benefit from tools and frameworks based on revDroid to provide better user experience when Android permissions are revoked by the user. In this section, we envision a variety of scenarios where revDroid could work.

2.4.1 Application Development

As described in Section 2.2, most application developers are not accustomed to handling SecurityExceptions gracefully because the Android development documentation before Android 6.0 does not explicitly instruct application developers to handle cases of permission revocation and application developers need time to learn the new documentation. This issue can be mitigated by providing developers a development tool based on the revDroid analysis framework. During the application development process, application developers can conveniently find out the cases where their code does not handle permission revocation correctly and fix the issues instantly based on the analysis report of the revDroid analysis framework. Moreover, this development tool can be further improved as a plug-in installed right into developers’ development environment. In this way, programming issues which may lead to application crash when permissions are revoked can be fixed before the application packages are shipped to application markets and end users.

2.4.2 Application Distribution

Application markets such as Google Play Store typically scan applications automatically when they are first published (e.g., Bouncer [26] scans for malicious applications). They can piggyback the test to detect potential crashes caused by permission revocation with the revDroid analysis framework. Then, application markets can notify developers if any potential issue is found, and application developers can upload a revised version to application markets for review. Thus application markets can guarantee that applications provided by them for downloading generally have a higher quality and better user experience, especially when they are running on devices allowing users to selectively grant or revoke permissions of applications.

3. ANALYSIS FRAMEWORK OF REV DROID

We define that a permission-relevant API call is not correctly handled if all methods in one of the call stacks of the API call are not wrapped by an exception handler or a proactive permission check block which calls permission check methods such as checkPermission and checkUidPermission. If the permission-relevant API call is not correctly handled, we define this usage as a mis-usage of the API.

In the analysis framework, revDroid firstly leverages apktool [6] to unpack the input APK file, and decode resource files including the manifest file and the UI XML files, to nearly original form [6]. After revDroid obtains the manifest file, it calculates the set of API calls which the application may invoke based on the permissions requested by the application and the result of PScout [8] which is a mapping between a permission and a set of API calls if the
permission is required on execution of the API\(^1\). Note that in Android 6.0, permissions which are classified as PROTECTION_NORMAL, including widely used permissions such as the INTERNET permission and the ACCESS_NETWORK_STATE permission will be automatically granted by system at install-time and cannot be revoked by users. Thus, we will ignore APIs which are only protected by permissions classified as PROTECTION_NORMAL in Android 6.0.

Secondly, revDroid leverages FlowDroid to generate a dummy main method and a dummy main method as the analysis entry point. The dummy main class and dummy main method is generated by combining the lifecycle methods of the application component classes, such as the void onCreate(Bundle) method of the android.app.Activity class. Moreover, the callback methods for system-event, handling UI interaction and others, including callback methods declared in the XML resource files, such as the callback methods of the callback classes and interfaces shown in Table 1, are also included in the dummy main method by parsing the decoded resource files [7].

Next, revDroid passes the generated dummy main class and dummy main method to Soot. Soot generates the whole call graph of the application, and starting from the dummy main method, examines bodies of every reachable method, looks for the method invocations which belong to the set of API calls calculated in the first step, and finally checks whether these API calls are wrapped in an exception handler which handles SecurityException or a proactive permission check block which calls permission check methods such as checkPermission and checkSelfPermission. If not, revDroid searches their callers, recursively. In one iteration of the recursive process, revDroid searches the caller of the method which current invocation locates at, checks all the invocations of the call stack, and moves to next iteration by treating every caller as callee if the invocation is not correctly handled. The search terminates when the analyzer cannot find any caller of the current callee or a method call cycle is found.

3.1 Mis-usage Detection Algorithm

The mis-usage detection algorithm shown in Algorithm 1 checks whether the API call is a mis-usage. The algorithm takes an API call as input, then leverages Soot to obtain all call stacks of this API call. For every call stack we check whether there exists an invocation in this call stack which is wrapped by an exception handler, or dealt with proactive permission check (described in detail in Section 3.2.2). If any one of the call stacks of the API call is neither wrapped by an exception handler nor dealt with proactive permission check, the API call is judged as a mis-usage. Note that, if the exception which is handled is a more general one than the SecurityException, i.e., super classes of SecurityException, we also consider that the developer correctly handles the invocation of the permission-relevant API.

Our algorithm checks all the reachable statements for the API call recursively, so the complexity of the algorithm depends on the number of statements and size of the call stacks. If there are \(N\) statements in the application and the average size of the call stacks is \(M\), the complexity of the algorithm is \(O(NM)\).

Algorithm 1 Mis-usage Detection

Require: APICall: a permission-relevant API usage to be checked
Ensure: checkResult: whether the usage of APICall is correct

1: checkResult ← true
2: for all callStack generate by Soot do
3: checkTempResult ← false
4: for all statement ∈ callStack do
5: if statement is wrapped by exception handler or 
dealt with proactive permission check then
6: checkTempResult = true
7: end if
8: end for
9: if checkTempResult == false then
10: checkResult = false
11: end if
12: end for

3.2 Technical Issues in the Framework

3.2.1 Recursion

revDroid checks whether the potential SecurityException is handled along a call stack in depth-first order. To prevent recursion from resulting in an infinite loop, we keep track of the history of invocations which have been checked along a call stack. If the invocation which should be checked next is already present in the history, we stop the check and judge the root invocation of the call stack as a mis-usage.

3.2.2 Proactive Permission Checks

Many applications, especially third-party libraries, will proactively check if the application is granted with the corresponding permissions by calling methods such as checkPermission, checkUidPermission and checkSelfPermission before invoking APIs which require permissions. revDroid takes this kind of handling into consideration by checking whether the API call depends on the return value of the permission check methods by leveraging the SimpleDominatorsFinder class of Soot which finds dominators of a given flow graph using the simple LT algorithm [15]. If that is the case, revDroid will judge these API calls as correctly handled.

3.2.3 Dead Code

Dead code is the code which exists in the application but will never be executed. Not considering the effect of the dead code will lead to false positives, so revDroid leverages
Soot to eliminate dead code first and only analyzes methods which are reachable [12].

4. EXPERIMENTS

4.1 Collections of Android Applications

The experiments are based on two datasets. The first dataset consists of 540 regular Android applications. We denote this as regular applications. This dataset is obtained by downloading the top 540 applications on the top free chart of the US Google Play Store in late September 2015, four months after Android 6.0 was released to developers in May 28th, 2015 [24]. The second dataset consists of 200 malware samples from the M0Droid project by Damshenas et al., which was last updated in December 2014 [18]. The experiments are conducted on a virtual machine with 4-core CPU and 12GB RAM running CentOS 6.5 and OpenJDK 7.

4.2 Results

We run revDroid with the two datasets mentioned in Section 4.1 and obtain a preliminary report. For each application, the preliminary report shows the permissions which the application requests and potential SecurityExceptions which the application handles and does not handle. Further, we analyze the preliminary report and obtain more important and interesting insights.

Table 2 lists the top 10 most used APIs in the two datasets. We can see that the top 10 most used APIs in the two datasets almost overlap, so applications’ usages of the applications in the two datasets are similar.

4.2.1 Results by Application

Figure 1 shows the overall distribution of regular applications’ handling of potential side effects. Of the 540 applications in the regular application dataset, revDroid analyzes 420 applications successfully. Among them, 172 applications do not call APIs which require permissions except permissions classified as PROTECTION_NORMAL in Android 6.0. For the rest 248 regular applications, 70% of them fail to handle one or more cases where the exception is supposed to be thrown. Only 30% of them handle all cases. This low rate of correct exception handling indicates that application developers worrisomely fail to handle potential side effects of permission revocations. Figure 2 shows the distribution of regular applications’ handling of potential side effects categorized by the rate of handled potential side effects. We can see that 48% of applications handle less than 60% of their potential side effects.

By contrast, Figure 3 and Figure 4 show the overall distribution of malwares’ handling of potential side effects. Of the 200 applications in the malware dataset, revDroid analyzes 155 malwares successfully. 23 malwares do not request permissions except permissions classified as PROTECTION_NORMAL in Android 6.0. We can conclude from the comparison between Figure 2 and Figure 4 that malwares put more attention to the side effects of permission revocation because 73% malwares handle all potential side effects.

A possible reason for this difference between the results for regular applications and malwares is that the developers of malwares tend to take end users’ revocation of permissions into consideration and handle potential side effects of permission revocation so that their malwares can survive on more devices, including those which allow users to revoke permissions, to keep the core malicious logic running. As we discover in Section 4.2.6, when trying retrieving the unique identifier string of the device, some malwares will return string “invalid” instead if the READ_PHONE_STATE permission is revoked to prevent itself from crashing.

4.2.2 Results by API

We group the results by the API calls which are invoked by the applications. Figure 5 shows the overall distribution of regular applications’ handling of potential side effects by API calls. 46% of API calls which require permissions are not correctly handled by regular applications, while the rest 54% API calls are correctly handled. Table 3 lists the top 10 most mis-used APIs by regular application developers. We can conclude from Table 3 that the API calls which require the ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION permissions are widely used by application developers but not handled well, which is also supported by the result shown in Table 5.

Since Android 6.0 allows users to revoke permissions after installing applications, we wonder if removing permissions will still crash applications. To answer this question, we developed a test application that invokes APIs that request permissions including those listed in Table 3, without
Table 2: Top 10 most used APIs of applications in the two datasets

<table>
<thead>
<tr>
<th>Rank</th>
<th>Regular Application</th>
<th>Malware</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>android.telephony.TelephonyManager: java.lang.String getDeviceId()</td>
<td>android.telephony.TelephonyManager: java.lang.String getDeviceId()</td>
</tr>
<tr>
<td>2</td>
<td>android.location.LocationManager: android.location.Location getLastKnownLocation(java.lang.String)</td>
<td>android.location.LocationManager: android.location.Location getLastKnownLocation(java.lang.String)</td>
</tr>
<tr>
<td>3</td>
<td>android.location.LocationManager: boolean isProviderEnabled(java.lang.String)</td>
<td>android.location.LocationManager: java.lang.String getBestProvider(android.location.Criteria,boolean)</td>
</tr>
<tr>
<td>5</td>
<td>android.location.LocationManager: void requestLocationUpdates(java.lang.String,long,float,android.location.LocationListener)</td>
<td>android.location.LocationManager: void requestLocationUpdates(java.lang.String,long,float,android.location.LocationListener)</td>
</tr>
<tr>
<td>6</td>
<td>android.location.LocationManager: android.location.LocationProvider getProvider(java.lang.String)</td>
<td>android.accounts.AccountManager: android.accounts.Account[] getAccounts()</td>
</tr>
<tr>
<td>8</td>
<td>android.hardware.Camera: android.hardware.Camera open()</td>
<td>android.telephony.TelephonyManager: java.lang.String getSubscriberId()</td>
</tr>
<tr>
<td>9</td>
<td>android.telephony.TelephonyManager: java.lang.String getSubscriberId()</td>
<td>android.location.LocationManager: boolean isProviderEnabled(java.lang.String)</td>
</tr>
</tbody>
</table>

Table 3: Top 10 most mis-used APIs of regular applications

<table>
<thead>
<tr>
<th>API</th>
<th>Permission</th>
<th>Mis-usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.location.LocationManager: android.location.Location getLastKnownLocation (java.lang.String)</td>
<td>ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION</td>
<td>77</td>
</tr>
<tr>
<td>android.accounts.AccountManager: android.accounts.Account[] getAccountsByType(java.lang.String)</td>
<td>GET_ACCOUNTS</td>
<td>76</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager: java.lang.String getDeviceId()</td>
<td>READ_PHONE_STATE</td>
<td>70</td>
</tr>
<tr>
<td>android.location.LocationManager: boolean isProviderEnabled(java.lang.String)</td>
<td>ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION</td>
<td>58</td>
</tr>
<tr>
<td>android.location.LocationManager: void requestLocationUpdates(java.lang.String,long,float,android.location.LocationListener)</td>
<td>ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION</td>
<td>43</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager: void listen(android.telephony.PhoneStateListener,int)</td>
<td>ACCESS_COARSE_LOCATION, READ_PHONE_STATE, ACCESS_FINE_LOCATION</td>
<td>42</td>
</tr>
<tr>
<td>android.location.LocationManager: android.location.LocationProvider.getProvider(java.lang.String)</td>
<td>ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION</td>
<td>36</td>
</tr>
<tr>
<td>android.accounts.AccountManager: android.accounts.Account[] getAccounts()</td>
<td>GET_ACCOUNTS</td>
<td>28</td>
</tr>
<tr>
<td>android.location.LocationManager: java.util.List getProviders(boolean)</td>
<td>ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION</td>
<td>27</td>
</tr>
</tbody>
</table>
Malwares handling all potential side effects, 96, 73%

Malwares not handling all potential side effects, 36 (27%)

Figure 3: Distribution of malwares’ handling of potential side effects. 27% of malwares do not handle all occurrences of potential side effects, while 73% of malwares handle all.

Malwares handling all potential side effects, 96, 73%

Malwares not handling all potential side effects, 36 (27%)

Figure 4: Distribution of malwares’ handling of potential side effects categorized by correct usage rate

Table 4: Examples of APIs which will cause crash even in Android 6.0 if not correct used

<table>
<thead>
<tr>
<th>API</th>
<th>Mis-usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.location.LocationManager: android.location.Location getLastKnownLocation(java.lang.String)</td>
<td>77</td>
</tr>
<tr>
<td>android.telephony.TelephonyManager: java.lang.String getDeviceId()</td>
<td>70</td>
</tr>
<tr>
<td>android.location.LocationManager: boolean isEnabled(java.lang.String)</td>
<td>58</td>
</tr>
<tr>
<td>android.location.LocationManager: android.location.LocationProvider getProvider(java.lang.String)</td>
<td>36</td>
</tr>
</tbody>
</table>

On the other hand, as shown in Figure 6, 36% API calls in malwares which require permissions are not correctly handled. The percentage is smaller than that of regular applications, which echoes the conclusion in Section 4.2.1 that developers of malwares handle potential side effects better than those of regular applications.

Figure 6: Distribution of malwares’ handling of potential side effects by API calls. 36% of API calls which require permissions are not correctly handled by malwares.

4.2.3 Results by Permission

For regular applications, we group the results by permissions to see how many occurrences of unhandled SecurityExceptions there are for API calls which require to work with one or more permissions. If an API call requires mul-
multiple permissions, the API call will be counted as the occurrences of mis-usages for multiple permissions.

Table 5 shows the correct usage rates for the permissions used by regular applications in our dataset. We can see from the table that except the permissions with low usage, the CAMERA permission has the highest correct usage rate of 94.83%. On the other hand, the GET_ACCOUNTS permission is frequently used but its correct usage rate of 24.35% is low.

The Android documentation and the code lint check functionality in the IDEs for Android [1] partially lead to the difference between the correct usage rates of different permissions. For example, we discover that in the API guides for the camera functionality, Google explicitly asks developers to always check for exceptions when using the camera and gives a code example which wraps the invocation of camera API with an exception handler [2]. In addition, the code lint check functionality which is provided by Google and built in the IDEs for Android, e.g., Android Studio and Eclipse, will show warnings to instruct developers to handle permission revocation for revocable permissions in Android 6.0 [1]. However, as we test in Android Studio, this functionality has two issues: i) this functionality is only available for Android Studio 1.4 or later and not provided on previous versions of Android Studio; ii) not all APIs which require revocable permissions in Android 6.0 are checked. For example, the code lint check functionality will warn developers about permission revocation for the android.location.Location getLastKnownLocation (java.lang.String) API, while will not warn developers about permission revocation for the android.accounts.AccountManager: android.accounts.Account[] getAccountsByType(java.lang.String) API.

4.2.4 Source of Mis-usages in Regular Applications

Many regular applications include third-party libraries, such as the AdMob ad library² and the Facebook SDK³, which are not written by the developers of the applications but share all the permissions with the hosting applications [13]. Third-party libraries tend to have common package name prefixes, like com.google.ads or com.facebook.android [16]. These libraries could unfortunately be the source of mis-usages, even when the libraries are written by Google itself, such as Google's Google Play Services⁴ which is an API package for Android applications providing access to a variety of Google services. Therefore, it is necessary to find out the actual source of mis-usages. We divide the mis-usages in regular applications into two categories: one is mis-usages which are caused by the application developers themselves, and the other is mis-usages which are caused by third-party libraries. Our set of third-party libraries contains 86 common libraries which are widely used by Android applications, especially popular Android applications on the Google Play Store. We distinguish the code from third-party libraries from developers’ own code by matching package name prefixes. The set of third-party libraries are further classified into the following four categories:

- **Ad libraries.** Ad libraries allow developers to embed ads into their applications to achieve monetization, for example, Google AdMob, StartApp⁵ and Tapjoy⁶.

- **Analytics libraries.** Analytics libraries help developers collect data about the usage of their applications, including running logs, performance metrics, crash report and users’ characteristic, for example, New Relic⁷, Splunk⁸ and HockeyApp⁹.

- **Social libraries.** Social libraries, such as Facebook SDK, provide developers with the ability to integrate social features, including third-party login, social sharing and user experience personalization, into their applications.

- **Other libraries.** This category of libraries include libraries which do not belong to any of the aforementioned three categories, including Android Support Library¹⁰, Apache Cordova¹¹ and Apache Thrift¹².

As is shown in Figure 7, in the regular application dataset, developers’ own code accounts for the large majority (96%) of occurrences of unhandled potential side effects, while third-party libraries account for only 4%. We can see that third-party libraries account for much less mis-usages than application developers’ own code. As for different categories of third-party libraries, analytics libraries and social libraries have no mis-usages, while ad libraries and other libraries both make up the 2% of mis-usages. Note that for ad libraries, after careful examination, we find out that the 16 (2%) mis-usages are false-positives, so actually ad libraries also have no mis-usages.

For each category, we also calculate the correct usage rate. The result is shown in Table 6. Note that for ad libraries, as we describe in the last paragraph, actually the correct usage rate for ad libraries is 100%. As the fifth row and sixth rows in the table show, third-party libraries as a whole handle 90.72% of potential side effects, while the case for developers’ own code is worse, with a correct usage rate of 46.07%.

In conclusion, third-party libraries handle potential side effects of permission revocation much better than developers’ own code. One possible explanation is that third-party libraries may be run inside a large number of hosting applications with diverse permissions so they should be more robust as we describe in Section 3.2.2. However, this result also shows the possible awful situation when the developed applications based on these third-party libraries run on the new platform of Android 6.0.

4.2.5 Type of Handled Exceptions

For regular applications which handle the potential side effects of permission revocation, we manually examine the type of exceptions which are handled. Most of the exceptions

---

²https://www.google.com/admob/
³https://developers.facebook.com/docs/android
⁴https://developers.google.com/android/guides/overview
⁵http://www.startapp.com/
⁶http://home.tapjoy.com/
⁷http://newrelic.com/
⁸http://www.splunk.com/
⁹http://hockeyapp.net/features/
¹¹https://cordova.apache.org/
¹²https://thrift.apache.org/
Table 5: Correct usage rates for permissions of regular applications

<table>
<thead>
<tr>
<th>Permission</th>
<th>Correct Usage</th>
<th>Mis-usage</th>
<th>Correct Usage Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BATTERY_STATS</td>
<td>3</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>CAMERA</td>
<td>55</td>
<td>3</td>
<td>94.83%</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>319</td>
<td>126</td>
<td>71.69%</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>5</td>
<td>2</td>
<td>71.43%</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>426</td>
<td>304</td>
<td>58.36%</td>
</tr>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>444</td>
<td>346</td>
<td>56.20%</td>
</tr>
<tr>
<td>RECORD_Audio</td>
<td>10</td>
<td>8</td>
<td>55.56%</td>
</tr>
<tr>
<td>WRITE_EXTERNAL_STORAGE</td>
<td>12</td>
<td>10</td>
<td>54.55%</td>
</tr>
<tr>
<td>GET_TASKS</td>
<td>24</td>
<td>21</td>
<td>53.33%</td>
</tr>
<tr>
<td>AUTHENTICATE_ACCOUNTS</td>
<td>50</td>
<td>49</td>
<td>50.51%</td>
</tr>
<tr>
<td>RESTART_PACKAGES</td>
<td>6</td>
<td>15</td>
<td>28.57%</td>
</tr>
<tr>
<td>USE_CREDENTIALS</td>
<td>12</td>
<td>30</td>
<td>28.57%</td>
</tr>
<tr>
<td>MANAGE_ACCOUNTS</td>
<td>14</td>
<td>43</td>
<td>24.56%</td>
</tr>
<tr>
<td>GET_ACCOUNTS</td>
<td>47</td>
<td>146</td>
<td>24.35%</td>
</tr>
<tr>
<td>READCALENDAR</td>
<td>2</td>
<td>29</td>
<td>6.45%</td>
</tr>
<tr>
<td>WRITE_SETTINGS</td>
<td>1</td>
<td>15</td>
<td>6.25%</td>
</tr>
<tr>
<td>READ_SOCIAL_STREAM</td>
<td>0</td>
<td>73</td>
<td>0.00%</td>
</tr>
<tr>
<td>READ_USER_DICTIONARY</td>
<td>0</td>
<td>72</td>
<td>0.00%</td>
</tr>
<tr>
<td>READ_CONTACTS</td>
<td>0</td>
<td>73</td>
<td>0.00%</td>
</tr>
<tr>
<td>WRITE_CONTACTS</td>
<td>0</td>
<td>73</td>
<td>0.00%</td>
</tr>
<tr>
<td>WRITECALENDAR</td>
<td>0</td>
<td>10</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Developers’ own code, 775 (96%)
Third-party libraries, 31 (4%)
Ad, 16 (2%)
Other, 15 (2%)

Figure 7: Distribution of unhandled potential side effects of regular applications. Third-party libraries account for much less occurrences of unhandled potential side effects than application developers’ own code.

Table 6: Correct usage rates for different categories of third-party libraries in regular applications. Third-party libraries’ code has a higher correct usage rate than developers’ own code.

<table>
<thead>
<tr>
<th>Category</th>
<th>Correct Usage</th>
<th>Mis-usage</th>
<th>Correct Usage Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad</td>
<td>150</td>
<td>16</td>
<td>90.36%</td>
</tr>
<tr>
<td>Analytics</td>
<td>110</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Social</td>
<td>4</td>
<td>0</td>
<td>100.00%</td>
</tr>
<tr>
<td>Other</td>
<td>39</td>
<td>15</td>
<td>72.22%</td>
</tr>
<tr>
<td>Third-party Libraries Subtotal</td>
<td>303</td>
<td>31</td>
<td>90.72%</td>
</tr>
<tr>
<td>Developers’ Own Code</td>
<td>662</td>
<td>775</td>
<td>46.97%</td>
</tr>
<tr>
<td>Total</td>
<td>965</td>
<td>806</td>
<td>54.39%</td>
</tr>
</tbody>
</table>
handled by applications are general RuntimeException, Exception, or even Throwable, which are super classes of SecurityException. Handling super classes of SecurityException can also prevent the side effects of permission revocation. The reasons why application developers handle the super classes of SecurityException instead of SecurityException are three-fold: i) developers need to handle other exceptions besides SecurityException. For example, when using the camera API, developers will need to handle not only permission revocation, but also cases where the camera is in use or does not exist, so all these cases are handled together with an exception handler for the general Exception class. ii) developers are intended to handle other exceptions, and SecurityException is handled incidentally. iii) information is lost when the application is reverse-engineered and Soot fails to infer the specific type of exceptions which are caught.

4.2.6 Content of Exception Handler

In addition, we also examine the content of the exception handler. The operations which applications execute in the exception handler can be divided into three categories. Firstly, most of the regular applications which catch the exception do not make anything meaningful in the exception handler, or even provide an empty exception handler. Secondly, there are applications which simply log the error or show a simple dialog informing the user that there occurs a failed permission check. Lastly, a small portion of applications will return fake data in the exception handler. For example, we discover that when trying retrieving the unique identifier string of the device, some families of malwares will return string “invalid” in the exception handler instead. In this way, these malwares can avoid crash caused by unhandled SecurityException, as well as NullPointerException, so their core malicious logic can keep running.

5. DISCUSSIONS

5.1 Limitations of Analysis Completeness and Success Rates

Native code: The completeness of the revDroid analysis result will be reduced if applications use native code. Using native code in Android applications will make it difficult to reverse-engineer the application. Our analysis tool depends on the Soot and FlowDroid frameworks which do not handle native modules. Because native code may contain mis-usage as well, the situation can be even more severe.

Limitations of Soot and FlowDroid: There are several other limitations in the Soot and FlowDroid frameworks which will lead to completeness issue and decrease the success rate of analysis. For example, Soot leverages SPARK algorithm to generate the call graph. To build a call graph edge, the algorithm must know the type of the base object on which the method is invoked. If the base object is null or comes out of a factory method inside the Android SDK, there will not be an edge from the base object. A typical example of this case in Android is the findViewById method of the android.app.Activity class. As a result, method call invoked on objects returned by the findViewById method will not be analyzed, which will lead to completeness issue. On the other hand, some applications in our dataset somehow contain byte-code which is not valid according to the specification of the Dalvik byte-code. For example, some APK files contain classes implementing interfaces which are not actually interfaces, and some other APK files contain inner classes located inside of an outer class, but the outer class itself is missing. Soot and FlowDroid will fail to analyze or instrument these applications completely or even crash. We will assist the maintainers of Soot and FlowDroid with handling these strange cases in the future.

5.2 Limitations of Our Measure on Side Effects

Although the code analysis of the side effects after permission revocation can cover more cases of users’ mis-usage of permission-relevant APIs, the mis-behavior of applications caused by the exception handler, is not measured in our analysis tool. When we look into the content of exception handlers, many developers simply catch the SecurityException or general Exception but do nothing else. This could explain why sometimes an application hangs rather than crashes after the permission is revoked. Moreover, we do not differentiate exception handlers for SecurityException and the general Exception. As a result, the real situation where the developers consider the permission revocation during their developers should be more severe.

5.3 Accuracy of Third-party Library Matching

As we described in Section 4.2.4, we use package name prefix matching to identify third-party libraries. Since application developers can specify package names for their code arbitrarily, which means application developers can even assign package names of other third-party libraries to his or her own applications, this third-party library matching approach may not be very accurate. In the future, we can leverage application similarity detection techniques such as AnDarwin [17] to mitigate this issue.

6. RELATED WORK

The issue of over-claim of permissions is one of the most popular security issues of Android [20]. Thus, many researchers [31, 11, 28] contributed a number of studies to analyze and mitigate this issue.

Barrera et al. analyzed the permissions requested by 1,100 free Android applications to investigate how the Android permission mechanism was used in practice and to determine its strengths and weaknesses [9]. Felt et al. performed a case study on Android platform by reviewing the top free and top paid applications from 18 Google Play categories [22]. For each of the reviewed applications, Felt et al. compared its functionalities with the permissions which it requested by manually exercising the user interface. The result showed that four out of 36 applications were over-privileged, and unnecessary INTERNET permission accounted for three of the over-privileged applications. Moreover, Felt et al. built Stowaway, an automatic tool to detect over-claim of permissions in Android applications [21]. Felt et al. applied automated testing techniques to Android 2.2 to determine the permissions required to invoke each API method and leveraged Stowaway to analyze a set of 940 applications. About one-third of these applications were identified to have unnecessary permissions. Wei et al. also applied Stowaway to a set of 237 evolving third-party applications covering 1,703 versions and found that the overall tendency was towards over-claim of permission [36].
To mitigate the issue of permission over-claim, Nauman et al. proposed Apex, which allowed users to grant or revoke a subset of permissions requested by the application using a simple and easy-to-use interface provided by an augmented application installer [31]. Similarly, Zhou et al. and Mueller et al. also developed TISSA [39] and Flex-P [30] respectively to provide finer-grained permission administration tools for Android.

However, in the meantime some of these countermeasures led to the side effect of applications crashing when one or more permissions was revoked. Kennedy et al. quantitatively measured the effects of removing permissions from Android applications by developing Pyandrazzi, a system for automated testing and measurement of the fatal exception behaviors [29]. Our work differs from theirs in that we detect application crash caused by permission revocation before the application is installed. Thus both users and developers can be notified whether the application will crash with certain permission revoked.

Static analysis for Android application usually involves reverse-engineering APK files and doing analysis without actually running the applications. This method has been widely adopted by researchers. Batyuk et al. designed a static analysis service which allowed users to gain deep insight into applications’ internals including a list of included third-party advertising and analytics libraries, potential privacy leaks and native executable usage [10]. Wei et al. analyzed APK files statically to identify permissions that applications requested and identify intents, i.e., indirect resource access via deputy applications [37]. Wang et al. decompiled Android applications to analyze their program logic related to the mobile channels and showed that the lack of origin-based protection opened the door to a wide spectrum of cross-origin attacks [35]. These works share similar methodologies with ours but our goals are different.

7. CONCLUSION AND FUTURE WORK

In this paper, we build an automatic tool, revDroid, to analyze the potential side effects of permission revocation on both popular applications from Google Play Store and malwares. The results show that only 30% of regular applications from Google Play Store handle all potential side effects and only 54% of occurrences of potential side effects are handled. In addition, 73% of malwares handle all potential side effects and 64% of occurrences of potential side effects are handled. Thirdly, third-party libraries occupy only 4% of all the mis-usages in the regular application dataset and have a higher correct usage rate than applications’ original developer. According to these results, we can conclude that: i) when the Android 6.0 introduces the new permission mechanism which supports dynamic permission revocation, the ecosystem of Android applications is unprepared to handle this new technique; ii) malwares have higher chances to survive in dynamic permission revocation; iii) although the third-party libraries deal with the permission revocation better than regular applications, they could still be improved by their developers, including Google itself.

In our further work, we will enlarge our experiments, and implement our solution as a web service which can detect the mis-usages of permission revocation in APK files. In addition, we will analyze the reason of application crash when the platform of Android adopt some optimization methods, e.g., fake data, to mitigate the side effects. Last but not least, we will optimize and extend revDroid to support to automatically patch Android applications (APK files) without the support of developers.

Acknowledgment

This paper is supported by NSFC (Grant No. 61572136). The authors would like to thank anonymous reviewers for their comments.

8. REFERENCES


