Journal of High Speed Networks 22 (2016) 55–64
DOI 10.3233/JHS-160538
IOS Press

Automatic permission inference for hybrid mobile apps

Jian Mao [a,*], Hanjun Ma [a], Yue Chen [a], Yaoqi Jia [b] and Zhenkai Liang [b]

[a] *School of Electronic and Information Engineering, Beihang University, Beijing, China*

[b] *School of Computing, National University of Singapore, Singapore, Singapore*

Abstract. The application permission system is one of the key components of Android security. Developers often use it incorrectly and claim more permissions than necessary, due to limitations of developers' knowledge and development tools. When application's vulnerabilities are exploited, the additional permissions give attackers more capability to carry out attacks. Hybrid mobile applications (apps) are a class of mobile apps that are built from web technologies, such as HTML, JavaScript, and CSS. In such applications, it is often easier to inject third-party code through vulnerabilities. When developers do not specify app's permissions correctly, the injected code will result in dangerous actions breaching system security. In this paper, we develop an automatic tool to assist developers to identify the permissions required by the apps based on the hybrid mobile apps' runtime permission checking.

Keywords: Hybrid mobile app, permission, Android, security, automatic tool

1. Introduction

In recent years, Internet technology on mobile devices has been developed rapidly. Android [1] system is a popular open-source platform for mobile devices. To secure users' private data, Android system creates a sandbox to isolate apps, and adopts a permission system to constrain apps' capability. An app must declare the permissions it requires during its installation phase. It cannot be installed if the user does not agree. Furthermore, the installed app can only access the system resources that is defined by the permissions granted by users.

Though the permission system restricts the privileges of Android apps, developers often declare more permissions than necessary due to their limitations in understanding the permission system, as well as limitations in development tools. If over-privileged apps have vulnerabilities that can be exploited, attackers may misuse the extra permissions of the apps.

This problem has become more serious on an emerging type of apps, i.e. *hybrid mobile apps* [5]. Hybrid mobile apps are developed from web technologies, including HTML, JavaScript, and CSS. They receive popularity due to the support of cross-platform development. Though they benefit developers in cross-platform support, the common code-injection vulnerabilities are carried over to the Android platform. If security-sensitive permissions are authorized to hybrid mobile apps unnecessarily, code injected into these apps will compromise users' data security and privacy. Hybrid mobile apps are usually based on a hybrid framework, e.g., PhoneGap [17], which provides APIs for apps to access system resources. Unfortunately, we cannot rely on developers to specify the required permissions accurately without sufficient knowledge of the run-time API-to-Permission mapping. For example, the PhoneGap framework provides a coarse-grained default API feature-permission mapping, which converts the platform features specified by developers, such as Camera, into permissions of the app. Once a feature is indicated by the developer, PhoneGap will include *all* permissions related to the feature. This process also introduces the risk of over privilege. To restrict the necessary permission set and ensure the least privilege principle, it is necessary to have an approach that infers permissions required by an app.

[*]Corresponding author. E-mail: maojian@buaa.edu.cn.

Our observation. Before an app is released, it needs to go through extensive testing. During the testing process, most of the functionalities of the app will be covered. Therefore, the permissions required by the app execution will be triggered thoroughly. Monitoring apps' dynamic permission checking and analyzing the corresponding run-time permissions will provide sufficient information to infer the necessary permissions for the apps and reduce the risk of over-privilege.

Solution. In this paper, we propose a dynamic solution to identify over-claimed permissions by hybrid mobile apps. Our approach, called *MinPerm*, serves as the testing environment of an app, and identifies the minimal set of permissions required by the app. The key to our approach is an underlying framework, which dynamically extracts the necessary permissions of hybrid mobile apps during their execution. Our approach identifies the over-claimed permissions by comparing the extracted permissions with the permissions declared by the developer. We demonstrate the effectiveness of MinPerm, and also analyze the permission usage of popular hybrid mobile apps.

In summary, we make the following contributions in this paper.

- We study the over-privileged problems in hybrid mobile applications on the Android platform. We also study the permission checking system of Android. We propose a dynamic solution to identify the minimum set of the permission required by the hybrid mobile apps and filter out the corresponding over-claimed permissions.
- We prototype our solution on CyanogenMod 10 [7] by modifying the Android framework. The evaluation results demonstrate the effectiveness of our approach.

Paper organization. The rest of the paper is organized as follows. Section 2 introduces the background and related work of this project. Section 3 describes the design and implementation of MinPerm. Sections 4 and 5 discusses the results of our experimentation. The paper is concluded by Section 6.

2. Background and related work

2.1. Android permission system

Android platform provides a *permission-based* mechanism to enforce restrictions on applications' access to sensitive resources or privileged operations [18]. For example, to read a user's contact information in her/his device, an application must have the READ_CONTACTS permission. To get specific permissions, application developers are required to declare the permissions in the manifest files of their applications. For instance, an app can declare the READ_CONTACTS permission with <*uses-permission*> tags in *AndroidManifest.xml* as in Listing 1.

When a user installs an application which can be downloaded from Google Play [11] or third-party markets, e.g., WanDouJia [19], Android will ask for the user's consent on the application's declared permissions. Currently, Android prompts with all the declared permissions to the user before an app's installation. Once the user agrees to install an app, the installed app can invoke the Android APIs [2] to access resources corresponding to the approved permissions.

Android permission system tries to help users pay attention to the app's permission declaration, and understand the security implications of installing the app. However, according to previous studies, researchers have found that

extendedchars

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.myapp" >
<uses-permission android:name="android.permission.READ_CONTACTS" />
...
</manifest>
```

Listing 1. READ_CONTACTS permission declaration.

few people paid attention to the declared permissions during installation, and people's comprehension of permissions is also poor [9]. Since few Android users are aware of the declared permissions, inexperienced developers may declare more permissions than the apps need and introduce the over-privileged risk to the system.

2.2. *Hybrid mobile apps*

A recent development enables web technologies to be used to develop Android applications, which are called hybrid mobile apps. The hybrid frameworks (e.g., PhoneGap [17]) support developing hybrid mobile apps that run on nearly all widely used mobile operating systems, such as Android, iOS, and Windows Phone. With these hybrid frameworks, programmers can develop hybrid mobile apps using web languages, and achieve the same functionality as native Android apps written in Java [13].

Figure 1 depicts the general architecture of hybrid mobile apps. A hybrid mobile application contains two components, the app's logic written in web language (e.g., HTML, CSS and JavaScript), and JavaScript-to-native interfaces provided by hybrid frameworks written in Java. When hybrid mobile apps call basic JavaScript functions (e.g., `alert` or `getElementById`) in web application code, it results in corresponding events triggered in the WebView component of Android. However, when the apps call JavaScript functions that are used to access native resources such as Camera and Contact, certain functions in WebView [20] will call JavaScript-to-native APIs. After the hybrid frameworks receive this call, it will access the Application Frameworks in Android OS, and the Frameworks will call the native function, access the native resource and return the result.

2.3. *Android permission checking system*

Android permission system provides a mechanism to check whether the app has certain permission as Fig. 2 shows. When an app accesses some resources of the system, it will call the system APIs first and then the APIs will call the permission checking function. The most basic function of checking permission is called *checkPermission*. The function is in charge of checking whether a permission has been declared for the app. If the app has declared the permission while its installation, the function will indicate a successful result by returning a true value. If the app has not declared the permission, the permission checking function will return a value indicating failure and show an execution error to the system and the user.

2.4. *Related work*

Android permission. Stowaway [8] determines the set of APIs and the corresponding permissions that an app uses. Stowaway finds out that one-third of 940 apps are over-privileged. However, Stowaway only evaluates the

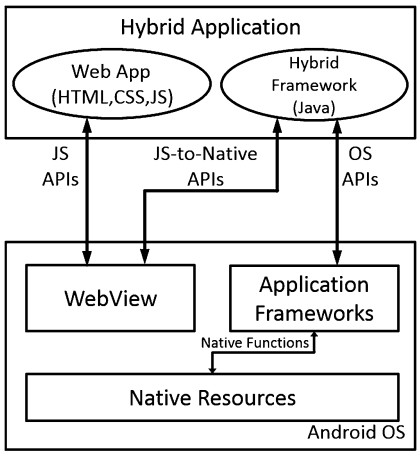

Fig. 1. General architecture of hybrid mobile apps.

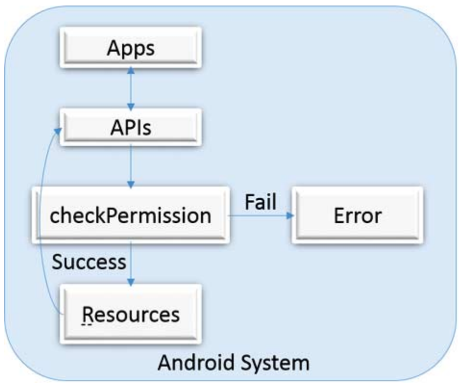

Fig. 2. Flow of permission checking.

Java code in native Android apps, but ignored the JavaScript code in hybrid mobile apps. Another study conducted by Felt et al. shows that only 17% of participants paid attention to permissions during installation, and just 3% of them could correctly answer all three permission comprehension questions [9]. From these we can see that most Android users do not pay much attention to what permissions the apps have and actually need.

Attacks on hybrid mobile apps. Recent works show that WebView and hybrid mobile apps (e.g., PhoneGap-based apps) are vulnerable to various attacks [6,10,14,16]. Luo et al. discover that malicious JavaScript code can perform privileged operations via the *addJavascriptinterface* [16]. Jin et al. introduce code injection attacks on HTML5-based mobile apps via internal and external channels [14]. By injecting malicious JavaScript into HTML5-based apps, the attacker can leak or pollute sensitive resources via the JavaScript-to-native interfaces on behalf of the user.

HTML5 security mechanism. Du et al. [15] study the risks of HTML5 apps and the drawback of access control in HTML5 system. They propose a fine-grained access control mechanism for the Android system in which they enforce the *effective permission* to be the intersection of the *frame permission*, *origin permission* and the *app permission*. Gerorgiev et al. [10] analyze the framework of the hybrid mobile apps and find some generic vulnerabilities. They study the fracking vulnerabilities in hybrid mobile apps and develop a capability-based model called NOFRAK to defend against the fracking attacks. This permits the apps to load contents from the third party, but isolates these external contents from local resources.

3. Our approach: MinPerm

In this section, we give an overview of our approach, including extracting required permissions and declared permissions of hybrid mobile apps, and comparing declared permissions with required permissions to detect abnormal behaviors. Then we describe the challenges of our approach in design.

3.1. Overview

Figure 3 shows the system components for execution and analysis of hybrid mobile apps in our approach. The highlighted components are the modules we created or the original Android components which we modified.

Our approach consists of four modules, the permission extraction module, the permission logging module, the permission identification module, and the permission comparision module.

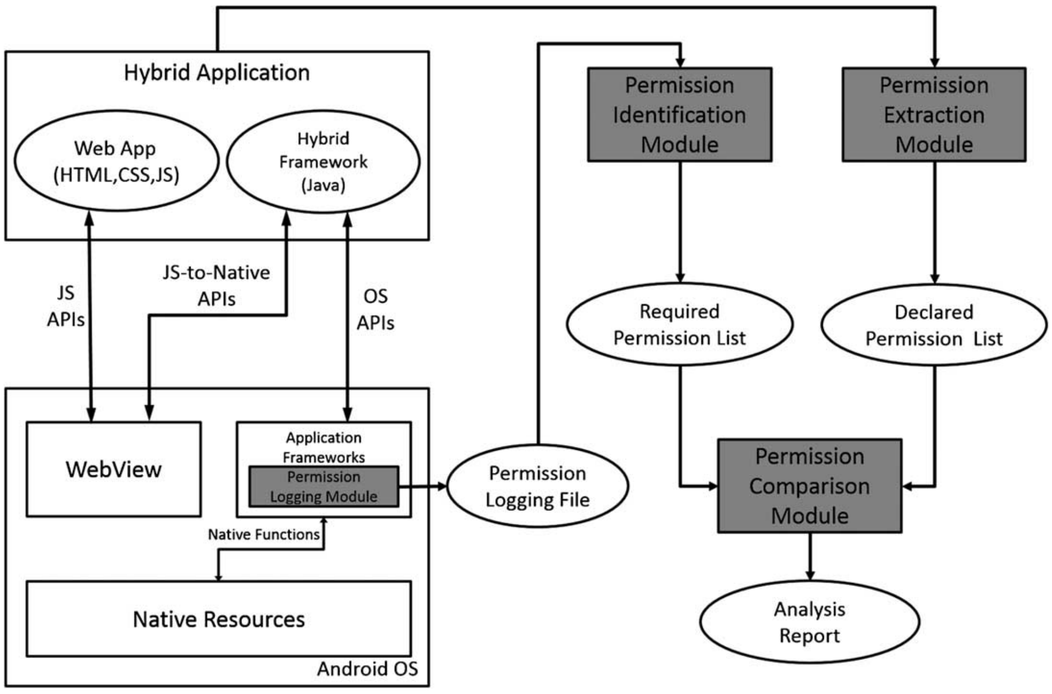

Fig. 3. The architecture of MinPerm.

The permission extraction module takes an app's apk file as the input and decodes it. Then it extracts the permission from the *AndroidManifest.xml* file of the decoded apk files, and writes the permissions into the declared permissions list. The permission extraction module can handle multiple apk files and extract their permission lists at the same time.

The permission logging module obtains the permission checking information while the apps are running, and writes to a log file. To obtain the permissions information, we hook the permission checking function under the frameworks in the Android source code. With these hooks, we can record the permission checking information when Android checks whether an app is granted with certain permissions.

The permission identification module reads the log file written by the permission logging module, identifies the permissions the app need, and sends it to the permission comparision module.

The permission comparision module takes the require permission file and the declare permission file as input. Then it formats these files into required permission list and declared permission list, and checks their difference. If there are permissions shown in the declared permission list, but not in the required permission list, then we can output the over-claimed permissions and provide a set of minimal permissions required during the testing process to the developer, so he or she can fix this app's permission over-declare problem.

3.2. Permission interference and verification

One of the main challenges to our approach is to modify the Android framework to extract the permissions needed by the app. We study and analyze the permission checking system in Android system and hook the function that checks permissions. When the app is running and needs to do some privileged operations, the Android system will call the permission checking function. So based on the hooks we added in the this function, we can get the permissions being checked, which is also the permissions needed by the app.

We have prototyped our solution on CyanogenMod 10 [7] by modifying the Android frameworks. Our main modification is in the component *Frameworks*. To build the permission logging module, we analyze the permission checking module which is inside the Frameworks component. We deploy hooks into the function inside the

Frameworks. This function, called *checkPermission*, is in charge of permission checking, and it is located at */frameworks/base/core/java/android/app/ContextImpl.java*. These hooks gather information such as what permissions are needed by the app, and send the permission information to the Logcat module and write it into a log file. At last the permission identification module reads the log file, extracts the log data, and writes it to the required permission list.

Although the apk file is just a zip package, since the XML files inside are encoded, we cannot directly read them after unzipping the apk file. The permission extraction module utilizes a decoded tool called *apktool* [3] to decode the apk file of an app and get a readable *AndroidManifest.xml* file. In the AndroidManifest.xml file, the declared permissions are attributes of the element *uses-permission*. After the decoding, the permission extraction module uses *python xml.etree.ElementTree* package to extract the tab *uses-permission* and its value, for example, the original line is *<uses-permission android:name="android.permission.CAMERA"/>*, and after extracting it becomes *android.permission.CAMERA*. Finally the module writes the permissions into a new file as the declared permission list.

4. Evaluation

To show the prevalence of over-privileged hybrid mobile apps, we have evaluated 34 hybrid mobile apps, each of which is downloaded more than 50,000 times in Google Play [11]. We found that 47% of them declare more permissions than necessary. We conducted our experiments on a Google Nexus 7 tablet, which is flashed with CyanogenMod [7] 10 system of Android 4.1.2.

4.1. Permission usage of hybrid mobile apps

The Android system has defined 151 standard permissions, and the developers are able to create customized permissions as well. Our approach mainly focuses on standard permissions, and developer-created permissions can be handled similarly as the standard permissions.

We have investigated 1519 hybrid mobile apps from Google Play. By extracting the declared permissions from the *AndroidManifest.xml* files in the apk files, we found 13,509 declared permissions from these 1519 hybrid mobile apps, including 11,892 declared permissions (about 88.03%) that belong to the Android standard permission set.

Among the 11,892 standard permissions declared by the apps, we demonstrate the top 10 most frequently declared permissions in Fig. 4.

4.2. Case study

We show a case study of MinPerm using an app named "Ijustapp" [12], which is a social networking app and downloaded more than 50,000 times in Google Play. Using this app, people can take photos and share them with others on the Internet. The screenshots of this application are shown in Fig. 5.

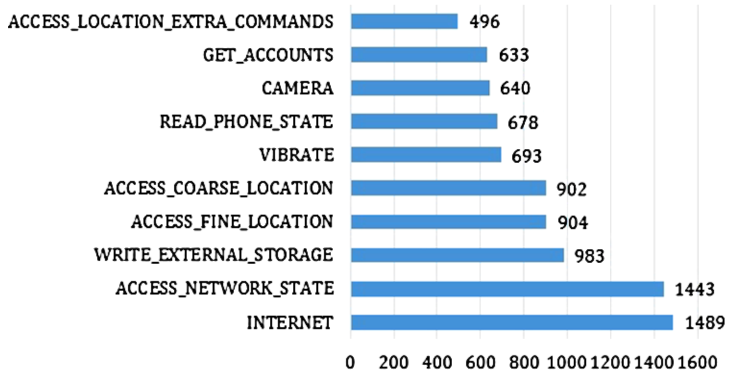

Fig. 4. Top 10 most frequently declared permissions.

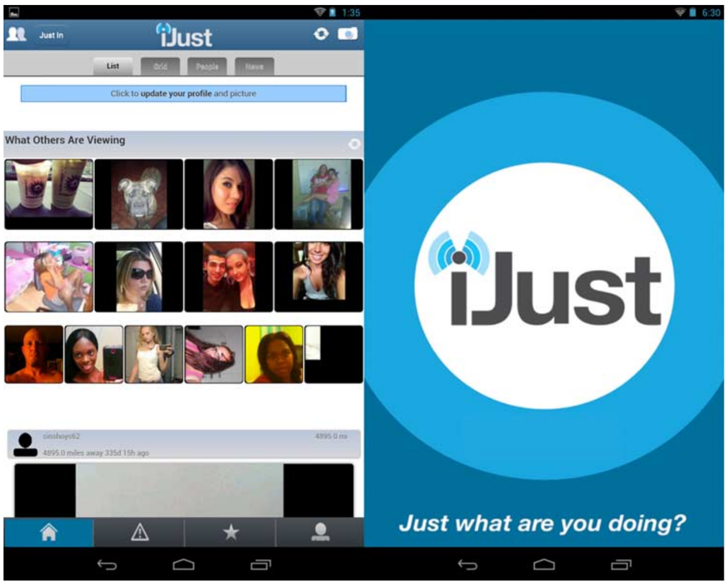

Fig. 5. Screenshots of Ijustapp.

To analyze this application, we run it with MinPerm. We manually trigger all the functionalities of the app. At the same time, MinPerm generates logs for this app, which contain the permissions needed by this app, and saves them to a permission log file. Then MinPerm moves on to analyze the log file. Firstly, MinPerm decodes the apk package of the app, extracts the permissions from the *AndroidManifest.xml* file, and writes them to the declared permissions file. Then MinPerm reads and formats the permission log file and the declared permissions file. Finally MinPerm compares these two files and check for overclaimed permissions of the app.

All the declared permissions we extracted from the AndroidManifest.xml files of this app are listed below:

- android.permission.
  ACCESS_COARSE_LOCATION
- android.permission.
  ACCESS_FINE_LOCATION
- android.permission.
  ACCESS_LOCATION_EXTRA_COMMANDS
- android.permission.
  ACCESS_NETWORK_STATE
- android.permission.
  BROADCAST_STICKY
- android.permission.
  CAMERA
- android.permission.
  GET_ACCOUNTS
- android.permission.
  INTERNET
- android.permission.
  READ_CONTACTS
- android.permission.
  READ_PHONE_STATE

- android.permission.
  VIBRATE
- android.permission.
  WRITE_CONTACTS
- android.permission.
  WRITE_EXTERNAL_STORAGE

With MinPerm, we found that “Ijustapp” over-claims 6 permissions, which are listed below.

- android.permission.
  ACCESS_COARSE_LOCATION
- android.permission.
  ACCESS_LOCATION_EXTRA_COMMANDS
- android.permission.
  BROADCAST_STICKY
- android.permission.
  CAMERA
- android.permission.
  READ_PHONE_STATE
- android.permission.
  WRITE_CONTACTS

Among them, “Ijustapp” declares three location permissions, while actually only *ACCESS_FINE_LOCATION* is needed, and *ACCESS_COARSE_LOCATION* and *ACCESS_LOCATION_EXTRA_COMMANDS* are over claimed. Second, it is often unnecessary for the hybrid mobile apps to claim the *CAMERA* permission, because they can call the system camera API without *CAMERA* permission. Finally, this app needs to obtain the contact information to search for friends on the Internet. But note that obtaining contact information does not need to write contacts, so the *WRITE_CONTACTS* permission is over claimed. With this permission over-claiming problem, if this app has some vulnerabilities, it may result in serious consequences. For example, attackers can modify or even delete all the contact information. To validate the result, we decoded this app, removed the over-declared permissions, re-compiled, and re-signed the app. By running the “trimmed” Ijustapp, we found that it works fine and the modification did not break the functionality of the app. With MinPerm, market operators or developers can detect the over-privileged apps, and refine them declaring proper permissions.

4.3. *Analysis of 34 apps*

The full result is shown in Table 1. The “Declared permissions” column shows the number of the permissions that each app declares, and the “Over-claimed permissions” column presents the number of the over-claimed permissions according to our analysis result. The result shows that 47% of hybrid mobile apps declare more permissions than needed. Moreover, the average number of the over-claimed permission is 1.03. This means one apps will have one more claimed permission on average. We also find the *WRITE_EXTERNAL_STORAGE* is the most over-claimed permission which represents 20% of the over-claiming.

5. Discussion

As shown in Section 4, MinPerm demonstrates that almost half of the popular hybrid mobile apps have over-claimed permissions. Based on the investigation and our experience, we found that hybrid mobile apps declare extra permissions because of the following reasons.

First, developers do not have a comprehensive understanding of the declared permissions and plugins. For example, due to the incomplete documentation in the early version of PhoneGap (Apache Cordova [4]), developers

Table 1
The summary of 34 hybrid mobile apps

App name	Declared permissions	Over-claimed permissions
accesvpn.version.v1	4	1
aming.app.NewsBook	9	2
app.cameraphotoeffects.com	7	4
app.devteam.vehiclesoundslite	4	0
app.ichwan.kamus.biologi	4	2
appinventor.ai_jpicer.PipefitterHandbook	2	1
appinventor.ai_p_tuttoilmondo.Fools	8	3
au.com.espace.dots.animal	4	0
au.com.espace.dots.xmas	4	0
au.com.espace.puzzle.alphaNum	4	0
au.com.espace.puzzle.xmas	4	0
au.com.ipvsolutions.connectTheDots	4	0
au.com.ipvsolutions.connectTheDots.dinosaurs	4	0
au.gov.asic.trackmyspend	5	2
com.appinmotion.artofwar	18	0
com.artofwar	12	0
com.cobaltsign.readysetholiday	12	1
com.fsch.truthordare	16	1
com.hexagon3d.mychinese	15	0
com.jsplash.apps.sightreadingimprover	14	0
com.mathopen.androidmath	19	0
com.nitobi.nfb	13	0
com.phonegap.LibraryAnywhere	25	3
com.phonegap.wordart	13	3
com.programmerworld.HRInterviewQuestionsLite	12	0
com.soffia.thealphabet	16	0
com.sweetsugar.gallerylock	23	2
com.tiichme.words	14	1
de.trinimon.calculator	12	0
hu.alchimedia.Hangman	13	1
Lightmaker.PBR	2	0
net.ijustapp.ijust	16	6
net.klier.blutdruck	4	2
sworkitapp.sworkit.com	2	0

directly copy&paste the permissions of all the official plugins from the documents into the *AndroidManifest.xml* file in their apps. Some of these apps declare more than 14 permissions that they do not really need. What is worse, some developers may add more plugins than they need. Second, the mappings between plugins and permissions in PhoneGap are coarse-granular. For example, when installing contact plugin, PhoneGap framework automatically adds *READ_CONTACTS*, *WRITE_CONTACTS*, and *GET_ACCOUNT* permissions into the AndroidManifest.xml file. However, reading or searching contacts only requires the *READ_CONTACTS* permission and the *GET_ACCOUNT* permission, not the other one. Thus PhoneGap itself can lead to over-privilege of hybrid mobile apps. Third, the rules of the permissions for hybrid mobile apps are sometimes in conflict with intuition. For example, when the PhoneGap call the Camera plugin, it just need the *WRITE_EXTERNAL_STORAGE* permission and does not need the *CAMERA* permission. The reason of this is because this plugin eventually calls the system camera app, and to call this system camera app does not require *CAMERA* permission.

6. Conclusion

The application permission system is one of the key components of Android security. Over-privileged hybrid mobile apps might be misused by attackers. Although developers are trying to declare their permissions precisely and follow the least privilege principle, some of them failed because of insufficient knowledge of the run-time API-to-Permission map. In this paper, we propose a dynamic solution to infer the permissions required by hybrid mobile apps. Our approach, called *MinPerm*, serves as the testing environment of an app extracting the dynamic permission needed and identifies the over-claimed permissions. We demonstrate the effectiveness of MinPerm, and show that part (47%) of 34 evaluated hybrid mobile apps have over-claimed permissions.

Acknowledgements

The authors thank anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (No. 61402029), the Beijing Natural Science Foundation (No. 4132056), the National 973 Project of China (No. 2012CB315905), the National Natural Science Foundation of China (Nos. 61170246, 376349), and the Beijing Natural Science Foundation (No. 4122024).

References

[1] Android, available at: https://www.android.com/.
[2] Android APIs, available at: http://developer.android.com/reference/packages.html.
[3] Android-apktool, available at: https://code.google.com/p/android-apktool/.
[4] Apache Cordova, available at: https://cordova.apache.org/.
[5] AppBuilder, What is a hybrid mobile app? 2012, available at: http://blogs.telerik.com/appbuilder/posts/12-06-14/what-is-a-hybrid-mobile-app.
[6] E. Chin and D. Wagner, Bifocals: Analyzing WebView vulnerabilities in Android applications, in: *Information Security Applications*, 2014, pp. 138–159.
[7] CyanogenMod, available at: http://www.cyanogenmod.org/.
[8] A.P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, Android permissions demystified, in: *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM, 2011, pp. 627–638.
[9] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin and D. Wagner, Android permissions: User attention, comprehension, and behavior, in: *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ACM, 2012, Article No. 3.
[10] M. Georgiev, S. Jana and V. Shmatikov, Breaking and fixing origin-based access control in hybrid web/mobile application frameworks, in: *NDSS*, 2014.
[11] Google Play, available at: http://play.google.com.
[12] Ijustapp, available at: https://play.google.com/store/apps/details?id=net.ijustapp.ijust.
[13] Java, What is Java?, available at: https://www.java.com/en/download/whatis_java.jsp.
[14] X. Jin, X. Hu, K. Ying, W. Du, H. Yin and G.N. Peri, Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation, in: *CCS*, 2014.
[15] X. Jin, L. Wang, T. Luo and W. Du, Fine-grained access control for HTML5-based mobile applications in Android, in: *Proceedings of the 16th Information Security Conference*, 2013.
[16] T. Luo, H. Hao, W. Du, Y. Wang and H. Yin, Attacks on WebView in the Android system, in: *ACSAC*, 2011.
[17] PhoneGap, available at: http://phonegap.com/.
[18] System permissions, available at: http://developer.android.com/guide/topics/security/permissions.html.
[19] WanDowJia, available at: http://www.wandoujia.com/apps.
[20] WebView, available at: http://developer.android.com/reference/android/webkit/WebView.html.