International Journal of Security and Its Applications
Vol. 11, No. 8 (2017), pp.59-76
http://dx.doi.org/10.14257/ijsia.2017.11.8.05

When Android Apps Open Ports to Handle Network Requests: Functionality or Security Vulnerability?

Hongzhou Yue[1] and Yuqing Zhang[1,2]

[1]State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an, China
[2]National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing, China
yuehz@nipc.org.cn, zhangyq@nipc.org.cn

Abstract

Large amounts of Android apps (applications) are found to open network ports to handle network requests to realize some specific functions, e.g., access from web page to Android app, communication between computer and Android device, file transmission in LAN (Local Area Network) environment, etc. However, an opened network port also provides an interface for attackers to visit the app. If a network request can trigger sensitive behaviors of a port-opening app without being effective authorized by the app, it would pose security threats to the user, and we consider this app has port-opening vulnerability.

In this paper, we first study the universality of port-opening apps in current Android app stores, the purposes of opening network ports and the possible attacks that the vulnerable apps may suffer from. Then we propose a detection method of port-opening vulnerability of Android app based on static analysis and implement a detection tool—APOVD (Android Port-Opening Vulnerability Detection). APOVD first judges whether an opened port can lead to the occurrence of sensitive behaviors by the method of reachability analysis and taint analysis. Then the technique of static program slicing is used to judge whether there exists adequate access controls in the paths to reach each sensitive behavior. If there exists a path to reach a sensitive behavior and no adequate access control in this path, APOVD considers that the app under test has port-opening vulnerability. 1187 port-opening Android apps are found in 15600 popular apps, and 407 of them are identified as having port-opening vulnerability with the help of APOVD. The result shows that APOVD is effective in detecting port-opening vulnerability.

Keywords: *Android Application; Port-Opening Vulnerability; Static Analysis; Reachability Analysis; Taint Analysis; Static Program Slicing*

1. Introduction

These years we have witnessed numerous security problems of Android apps that caused by opening interface to the outside. For instance, an exported component of an app can be visited by the other apps in the same device, which may lead to many types of vulnerabilities, e.g., capability leak [1,2], content provider leakage [3,4], privilege escalation [5,6], confused deputy [7], component hijacking [8,9], intent spoofing [10], etc. The use of WebView allows an app to expose registered interfaces to the JS (JavaScript) code of the web pages that displayed in the WebView, which may lead to attacks such as excess authorization, file-based cross-zone scripting, etc. [11,12]. These vulnerabilities have caused serious security problems to the users and attracted the attention of lots of researchers. However, none of these opening interface can be exploited remotely outside the device. Therefore, it leads to the difficulty of vulnerability exploitation, as the attacker must make efforts to let the vulnerability exploitation program run in the same device with the vulnerable app.

Recently, we notice a new type of vulnerability which is also caused by opening interface to the outside. But this time the opened interface is the network port, which may let the vulnerability not only be exploited locally, but also exploited remotely. Therefore, it is more dangerous than the vulnerabilities discovered before. We call this type of vulnerability the port-opening vulnerability. Baidu wormhole[1] is a typical case, which is caused by opening network ports of Android apps. Attackers can implement attacks by sending a remote or local request to the vulnerable app to steal private information, damage the file system, download and install malicious apps in the device, etc. It was reported that more than 100 million Android devices that with vulnerable apps installed had been affected by Baidu wormhole[2]. Except for Baidu wormhole, other security problems that caused by port-opening of Android apps were also reported by researchers[3].

In this paper, we focus on studying the phenomenon of opening network ports of Android apps and the detection method of port-opening vulnerability. The main contributions of this paper include:

(1) We first put forward the concept of port-opening vulnerability of Android app which has never been proposed and studied before, and we found by our study that it is a dangerous vulnerability and widely existed in the current Android apps.

(2) We do a statistical analysis work on 15600 popular Android apps to study the universality of port-opening Android apps, and the result shows that 1187 apps of them open network ports. An analysis work is also done to analyze the main purposes of opening network ports of these 1187 apps and the possible attacks that the vulnerable apps may suffer from.

(3) We propose a detection method of port-opening vulnerability based on static analysis and implement a detection tool—APOVD (Android Port-Opening Vulnerability Detection). APOVD first uses the method of reachability analysis and taint analysis to find whether an opened port can lead to the occurrence of sensitive behaviors. Then it judges whether there exists adequate access controls in the paths to reach each sensitive behavior by the method of program slicing. If there exists a path from a port-opening program point to a sensitive behavior in an Android app, and in this path there exists no access control or the access control is inadequate, then APOVD considers that there exists port-opening vulnerability in this app. 407 apps of the 1187 port-opening Android apps are found to have port-opening vulnerability by APOVD, and the result shows that APOVD is effective in detecting port-opening vulnerability.

The rest of this paper is organized as follows: We do some statistical analysis for the port-opening problem of Android app in Section 2. In Section 3, we give an introduction to our detection method of port-opening vulnerability of Android app and the implemented tool—APOVD. In Section 4, we show APOVD's performance in detecting port-opening vulnerability of Android app by experiment. We give some discussions about port-opening vulnerability and APOVD in Section 5. The related work is introduced in Section 6. In Section 7, we close the paper with conclusions.

2. Analysis of Port-Opening Android Apps

In this section, we discuss the universality of port-opening Android apps, the purposes of opening network ports and the security problems that may caused by the port-opening apps.

[1] https://www.enisa.europa.eu/publications/info-notes/baidus-moplus-sdk-wormhole
[2] http://thehackernews.com/2015/11/android-malware-backdoor.html
[3] http://blog.trendmicro.com/trendlabs-security-intelligence/open-socket-poses-risks-to-android-security-model/

2.1. Universality of Port-Opening Android Apps

To understand the universality of port-opening Android apps, we made a statistical analysis to the apps downloaded from Google Play[4] and Wandoujia (a famous Android app store in China[5]). The download time of the apps is from July 2016 to October 2016. The most popular apps were chosen to be download according to the download ranking of the app stores. There are 33 categories in Google Play and 18 categories in Wandoujia, and we downloaded 200 apps for each category in Google Play and 500 apps for each category in Wandoujia.

We applied the method of static scan and runtime verification to count the number of the apps that open network ports. First, the apps under test were processed by our static scan algorithm to find the apps that may open network ports, which was achieved by searching the feature APIs that can identify the behaviors of opening network ports of an app (e.g., the API "<*ServerSocket: accept()*>"). Then, the found apps were installed and run in the Android device to validate the opened ports. The result is shown in Table 1. From the result, we can see that there are 406 apps (among the 6600 apps) in Google Play and 781 apps (among the 9000 apps) in Wandoujia open network ports. The proportions of port-opening apps in Google Play and Wandoujia are 6.15% and 8.68% respectively, and they can be considered as a relatively high proportion. Besides, we also counted the numbers of installations (or downloads) of each app according to official figures released in the corresponding websites. The proportions of installations of the port-opening apps to total installations for Google Play and Wandoujia are 12.38% and 15.64% respectively. It means that these port-opening apps are very popular and have a large number of users. Therefore, once an app has port-opening vulnerability, it may cause widespread influence.

We also made a statistical analysis for these apps based on app category, and the result is shown in Table 2. We can see from Table 2 that categories with the largest percentage of port-opening apps in Google Play and Wandoujia are Tools and Video respectively. It can also be seen that categories with large percentage of port-opening apps have some similarities in Google Play and Wandoujia. The port-opening apps are mainly concentrated in the app types of tool, social, video, music, etc.

Table 1. Statistics of Port-opening Apps

APP Store	App Count	Category Count	Port-Opening App	
			Count (Ratio)	Installation Ratio
Google Play	6600	33	406 (6.15%)	12.38%
Wandoujia	9000	18	781 (8.68%)	15.64%

2.2. Purposes of Opening Network Ports

In order to understand the reasons why Android apps open network ports, we did a static analysis manually for some of the port-opening apps. According to our analysis, we found that Android apps open network ports mainly for the following purposes:

Interaction Between Web Page and Android App. We found that many app vendors open network ports in their developed Android apps for the purpose of interaction between web page and Android app. This type of Android app implements the port-opening by Http protocol, therefore the Http response sent from an app can be easily processed by the JS code in the web page. By this way, an app can provide necessary information of the device or perform some specific functions for the web page by the opened ports. The typical application scenarios include information collection, advertising push, app installation and upgrading, etc.

[4] https://play.google.com/store
[5] https://www.wandoujia.com/

Besides, according to our rough statistics, interaction between web page and Android app is the main purpose that Android apps open network ports. We also found that there are more port-opening apps in Wandoujia than Google Play (look at the data shown in Table 1). The main reason is that there are more apps in Wandoujia open network ports for this purpose than Google Play.

Table 2. The 10 Categories with the Largest Percentage of Port-opening Apps for Google Play and Wandoujia

Google Play			Wandoujia		
Category	**App Count**	**Port-Opening Count (Ratio)**	**Category**	**App Count**	**Port-Opening Count (Ratio)**
Tools	200	35 (17.5%)	Video	500	90 (18.0%)
Social	200	30 (15.0%)	System Tool	500	78 (15.6%)
Entertainment	200	29 (14.5%)	Chating & Social	500	71 (14.2%)
Music & Audio	200	26 (13.0%)	News & Reading	500	68 (13.6%)
Communications	200	26 (13.0%)	Music	500	62 (12.4%)
News & Magazines	200	24 (12.0%)	Life Services	500	55 (11.0%)
Shopping	200	23 (11.5%)	Shopping	500	55 (11.0%)
Sports	200	20 (10.0%)	Financial Management	500	49 (9.8%)
Maps & Navigation	200	20 (10.0%)	Sports & Healthy	500	45 (9.0%)
Finance	200	18 (9.0%)	Office efficiency	500	43 (8.6%)

PC Manages Android Device. These apps are used for managing Android device from the PC (personal computer) side. An app installed in the Android device opens a network port to receive administrative commands sent from computer, and perform the related operations in the Android device (e.g., manage the files, contacts, SMS records, etc.). There are two ways for a computer to connect with a mobile device—wired and wireless, both of which have been found the port-opening Android apps. The wired way is achieved by USB connecting, while the wireless way is achieved in the LAN environment (e.g., the wifi environment) where the computer and Android device share the same LAN segment.

File Transmission in LAN. File transmission in LAN not only includes file transmission between Android devices, but also file transmission between Android devices and other types of devices in the same LAN environment. An app installed in the Android device opens a network port to receive network requests from the other devices and provides the requested files as a response.

Communication in Android Application Layer. We noticed in the experiments that some apps open network ports just for the communication in Android application layer. The communication form includes the communication between apps in the same device, as well as the communication between the different components in the same app.

Others. Other typical purposes of opening network ports we have found include chatting in LAN, digital life, etc. In these cases, the purposes of opening network ports of Android apps include receiving messages sent from the other devices in the LAN environment, providing multimedia files outwards to achieve the wireless streaming media transmission between the different devices, etc.

Of course, the purposes are not limited to these mentioned above, they are just some of the purposes that we have successfully analysed and validated in experiments. We also cannot give the concrete numbers of apps for each purpose because the purpose analysis of opening network ports is very subjective and sometimes the purpose cannot be clearly identified.

2.3. Security Analysis of Port-Opening Android Apps

An Android app can provide necessary functions outwards by an opened network port, but it also provides an interface for the hackers to visit the app or device, which may pose security threats to the user. If the opened port can lead to the sensitive behaviors of an app (e.g., information leakage by the opened port, system damage, denial of service, etc.), and there is no adequate access control measure to restrict external access, then we consider that there exists port-opening vulnerability in this app.

In order to analyze the security of an Android app that has port-opening vulnerability, we assume that there is an port-opening app which can provide privacy information outwards by responding the network requests without any access control. Then we can give an attack model for this app, which is shown in Figure 1. If the Android device allows receiving remote network request (some device may refuse to accept the external network request), then the attacker can send remote request in the LAN environment to the device to get the privacy information. Besides, the attacker can also exploit the vulnerability locally using the following two methods: (1) The attacker lures the victim to download and install a malicious app. when the app runs, it sends request to the vulnerable app which is running in the same device, gets the privacy information and sends this information to a server in Internet which is controlled by the attacker. (2) If the opened port is implemented by Http protocol, then the attacker can create a malicious website, which contains web pages that have malicious JS code inside. Then the attacker lures the victim to visit these web pages. When users visit these web pages, the malicious JS program runs and sends request to the vulnerable app which is running in the same device. After receiving the response which contains privacy information, the JS program sends this information to a server in Internet which controlled by the attacker.

Of course, the possible attack methods are not limited to those shown in Figure 1, which is only used to show that the vulnerability exploitation methods are varied and easy to be implemented. Therefore it can be concluded that the port-opening vulnerability of Android app is very dangerous.

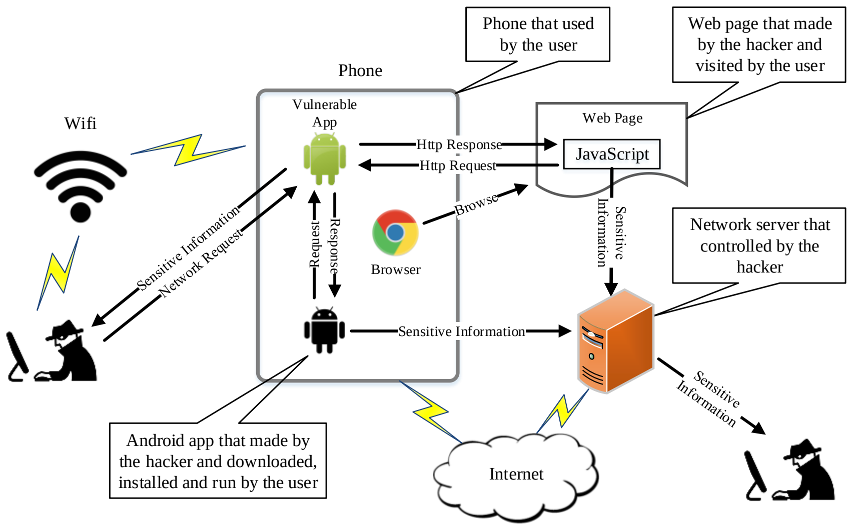

Phone that used by the user
Web page that made by the hacker and visited by the user
Phone
Wifi
Vulnerable App
Web Page
Http Response
Http Request
JavaScript
Browse
Sensitive Information
Network Request
Request
Response
Browser
Sensitive Information
Network server that controlled by the hacker
Sensitive Information
Sensitive Information
Android app that made by the hacker and downloaded, installed and run by the user
Internet

Figure 1. Attack Model of Port-opening Vulnerability

3. Detection of Port-Opening Vulnerability

In Section 2, we learned about the universality of port-opening Android apps, and we also recognized the danger of an app that has port-opening vulnerability. Therefore, it is necessary to find an effective method to detect port-opening vulnerability of Android app. In this section, we discuss the detection method of port-opening vulnerability.

3.1. Detection Method

In fact, the detection of port-opening vulnerability only needs to solve two problems—(1) Whether an opened port can lead to sensitive behaviors of an app. (2) Whether there exists effective access controls in the program paths to reach each sensitive behavior. We realized that dynamic analysis can hardly be used to detect port-opening vulnerability of Android app due to the demand for effective network input and the limitation of code coverage. Therefore, we use the method of static analysis to solve these two problems and apply our method to a detection tool—APOVD. The workflow of APOVD is shown in Figure 2, which mainly includes two parts—sensitive behavior detection and strength judgement of access control. These two parts are used to solve the two problems mentioned before respectively, and we will introduce their implementation details in Section 3.2 and 3.3.

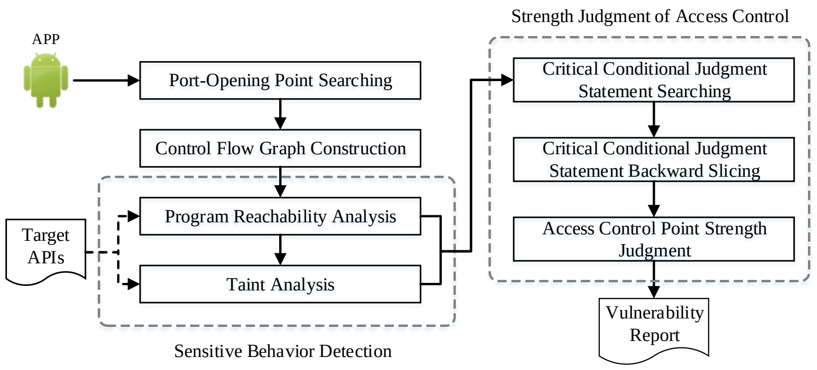

Strength Judgment of Access Control
APP
Port-Opening Point Searching
Control Flow Graph Construction
Program Reachability Analysis
Target APIs
Taint Analysis
Sensitive Behavior Detection
Critical Conditional Judgment Statement Searching
Critical Conditional Judgment Statement Backward Slicing
Access Control Point Strength Judgment
Vulnerability Report

Figure 2. Overview of APOVD

APOVD's static analysis is mainly based on Soot[6], a Java static analysis framework, and the target language of static analysis is Soot's intermediate representation—Jimple. The inputs of APOVD are the app under test and the target APIs file. The outputs of APOVD are the program paths to reach the sensitive behaviors and the results of strength judgment of access control.

3.2. Sensitive Behavior Detection

Construction of Control Flow Graph. In order to reduce the workload of static analysis, APOVD only considers the program that is related to the port-opening and builds the CFG around the port-opening program point. We define *NWP* (Network Waiting Point) as the program point at which the server side waits for the network input, and *NWP* can be characterized by some special API invocations. Listing 1 shows the four *NWP* APIs which are considered by APOVD. They involve two types of network protocols—TCP and UDP. *n1* is the *NWP* API corresponding to UDP, *n2* and *n3* are the *NWP* APIs corresponding to the two implementation ways of TCP—blocking sockets and non-blocking sockets. *n4* is the *NWP* API corresponding to the local socket of Android.

[6] Soot. URL: https://sable.github.io/soot/

n1:	<java.net.ServerSocket: java.net.Socket accept()>
n2:	<java.nio.channels.ServerSocketChannel: java.nio.channels.SocketChannel accept()>
n3:	<android.net.LocalSocket: android.net.LocalServerSocket accept()>
n3:	<java.net.DatagramSocket: void receive(java.net.DatagramPacket)>

Listing 1. Four types of *NWP*

APOVD first searches an app under test for the *NWPs*. If there exists *NWPs* in this app, then starts from the *NWPs*, APOVD reversely searches the method invocation relationship (which can be generated automatically by Soot) of the app to find the starting components that can lead to the invocation of these *NWPs*. Then depending on these starting components, APOVD generates a dummy main method for the app referring to the method proposed by Arzt, et al. [1]. In this dummy main method, the invocations to the lifecycle and callback methods in each starting component are constructed, taking into consideration the order of component lifecycles. Started from the dummy main method, APOVD constructs the CFG of the app based on Soot's CFG construction module. Besides, the results generated by EdgeMiner [13] are incorporated by APOVD to achieve a more complete CFG.

Definition of Sensitive Behavior. APOVD uses a feature API invocation sequence to define a program behavior. We use s to represent a sensitive behavior, and $s = \{A, R\}$. *A* is an API set and *R* is the invocation order of the APIs in *A* and their relationship. Each API *a* in *A* can be represented by a 4-tuple, i.e., $A=\{a\}=\{\langle api, in, out, para\rangle\}$. *api* represents the API name. *in* is used to identify whether the API needs to receive the network input. If it does, *in* is the location of the parameter. Otherwise, *in* is *null*. *out* is used to identify whether any content need to be output to the network as a network response after the API invocation. If it does, *out* is the content location. Otherwise, *out* is *null*. The content can be the input parameters or return value. *para* is used to identify whether the API needs specified parameters. If it does, *para* is represented by the form of $para=\{\langle pl, pv\rangle\}$, *pl* is the parameter location and *pv* is the concrete value of the parameter. Otherwise, *para* is *null*.

We use the examples shown in Listing 2 and 3 to illustrate the definition of sensitive behavior of APOVD. The result of sensitive behavior definition is shown in Table 3. Listing 2 shows the code fragment of two implementation methods of downloading file (line 1-4 and 6-9). The first method can be represented by API sequence of *(a1*, *a2*, *a3*), and the APIs should follow this invocation order, while the second method can be represented by API sequece of (*a1*, *a4*). *a1* needs to receive the network input as the first parameter. It means that only the url of the file downloaded is determined by the network request, the behavior of file downloading can be considered as a sensitive behavior. Listing 3 shows the code fragment of reading contacts. This behavior can be represented by API sequence of (*a5*, *a6*). The first parameter of *a5* must satisfy the form of "*content://com.android.contacts/**" (the symbol '*' means multiple matching, and 'Phone.*CONTENT URI*' satisfies this form), and the return value of *a6* needs to be output to the network as a network response after the API invocation (the number 0 represents the return value).

All the target APIs are defined in a text file, which is used as the input of APOVD (look at the "*Target APIs*" in Figure 2). Most of these APIs are obtained by our manual analysis work. The work results of Au, et al. [14] and Rasthofer, et al. [15] are also referenced to define some of the APIs. The APIs file can be easily extended by the user of APOVD by defining new sensitive behaviors.

```
URL url = new URL("http://filePath");
URLConnection conn = url.openConnection();
conn.connect();
InputStream is = conn.getInputStream();
…
DownloadManager dm =
(DownloadManager)getSystemService(Context.DOWNLOAD_SERVICE);
URL url = new URL("http://filePath");
DownloadManager.Request request = new Request(url);
long reference = dm.enqueue(request);
```

Listing 2. Code Example of Downloading File

```
ContentResolver resolver = mContext.getContentResolver();
Cursor pCursor = resolver.query(Phone.CONTENT_URI,
PHONES_PROJECTION , null, null, null);
if (pCursor != null) {
   while (pCursor.moveToNext()) {
      String phoneNumber =
      pCursor.getString(PHONES_NUMBER_INDEX);
…
}}
```

Listing 3. Code Example of Reading Contacts

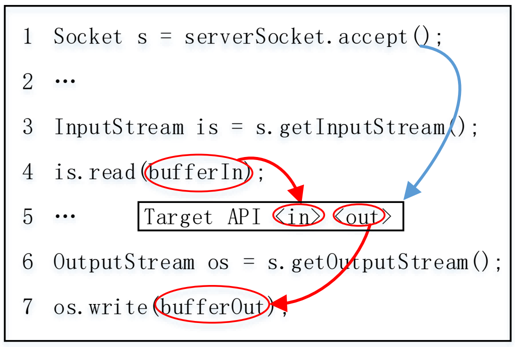
Table 3. Example of Sensitive Behavior Definition. "*N*" represents *null*.

Behavior	**API Set (*A*)**	**Ralation (*R*)**
Download file	*a1*: <URL: init, 1, *N*, *N*>	$(a1 \wedge a2 \wedge a3) \vee (a1 \wedge a4)$
	a2: <URL: openConnection, *N*, *N*, *N*>	
	a3: <URLConnection: getInputStream, *N*, *N*, *N*>	
	a4: <DownloadManager: enqueue, *N*, *N*, *N*>	
Read contacts	*a5*: <ContentResolver: query, *N*, *N*, {<1,*content://com.android.contacts/*>*}>	$a5 \wedge a6$
	a6: <Cursor: getString, *N*, 0, *N*>	

Reachability Analysis and Taint Analysis. Starts from *NWP*, APOVD processes reachability analysis along the CFG to judge whether the target API invocations corresponding to a sensitive behavior *s* can be reached. A constant propagation analysis is also done to judge whether *para* (*para* is not *null*) of each API *a* in *A* can be satisfied (e.g., the API *a5* in Table 3 must receive the specified value as the first parameter input). If the API invocations defined in *A* cannot be reached in the specified order defined by *R*, or there exists an API *a* in *A*, the *para* of which cannot be satisfied, behavior *s* would not be considered as being reached. Otherwise, APOVD processes taint analysis to further validate this sensitive behavior.

We difine *NDIP* (Network Data Input Point) as the program point at which the server side reads the network input, and the input buffer corresponding to *NDIP* is defined as *NDIB* (Network Data Input Buffer). *NDOP* (Network Data Output Point) is defined as the program point at which the server side writes the network output, and the output buffer corresponding to *NDOP* is defined as *NDOB* (Network Data Output Buffer). *NDIP* and *NDOP* are determined by *NWP*, and they are found by the reachability analysis of

APOVD. Take the code fragment shown in Figrue 3 as an example, statement in line 4 is the *NDIP* and the Java object *bufferIn* in line 4 is the *NDIB*. Statement in line 7 is the *NDOP* and the Java object *bufferOut* in line 7 is the *NDOB*.



```
1 Socket s = serverSocket.accept();
2 …
3 InputStream is = s.getInputStream();
4 is.read(bufferIn);
5 …          Target API <in> <out>
6 OutputStream os = s.getOutputStream();
7 os.write(bufferOut);
```

Figure 3. Reachability Analysis and Taint Analysis of APOVD

The blue solid arrow represents the reachability analysis and the red dashed arrow represents the taint analysis

APOVD mainly does two kinds of taint analysis. The first kind of taint analysis is to judge whether the input parameter corresponding to *in* (*in* is not null) of each API *a* in *A* can be tainted by *NDIB*, which is started from *NDIP* and terminated in API *a*. The second kind of taint analysis is to judge whether *NDOB* can be tainted by the content corresponding to *out* (*out* is not *null*) of each API *a* in *A*, which is started from API *a* and terminated in *NDOP*. The red dotted arrows in Figure 3 show the taint analysis paths of APOVD. APOVD's taint analysis is mainly based on FlowDroid [16] and has made some improvements on it.

After the taint analysis, if there exists *in* (*in* is not *null*) of API *a* cannot be tainted by *NDIB*, or there exists *out* (*out* is not *null*) of API *a* cannot taint *NDIB*, APOVD would consider that behavior *s* is not satisfied. Otherwise, behavior *s* would be considered as being reached.

3.3. Strength Judgment of Access Control

If there exists a sensitive behavior *s* can be reached, APOVD will judge the difficulty to reach *s* next. It is considered that port-opening vulnerability only exists in the context of no access control or inadequate access control, while an adequate access control cannot lead to the conclusion of port-opening vulnerability. The path from *NWP* to *s* is defined as the path from *NWP* to the first API invocation in the API sequence of *s*. It is considered that the access control only exists in some of the conditional judgment statements in the path from *NWP* to *s*. Therefore, APOVD first finds out all the conditional judgment statements that may be acted as access controls in the path from *NWP* to *s*, and then performs backward slicing on these conditional judgment statements. Finally, the generated program slices are used by APOVD to judge the strength of access control.

Critical Conditional Judgment Statement Searching. For a sensitive behavior *s*, if there exists a conditional judgment statement that must be passed through from *NWP* to *s*, and only one jump branch of it can lead to *s*, this conditional judgment statement is called *CCJS* (Critical Conditional Judgment Statement). Two kinds of conditional judgment statements are considered by APOVD—"*if*"and "*switch*" statement, and both of them can be acted as *CCJS*.

APOVD uses the method of reachability analysis to search the *CCJSs*. It can be considered that the path from *NWP* to *s* consists of a number of sub paths, and each sub path represents a running trace of the program. APOVD traverses all the sub paths from

NWP to *s* based on a depth-first algorithm. After the process of the depth-first traversal, all the *CCJSs* can be found in the path from *NWP* to *s*, represented as C_s.

Figure 4 shows the code structure of an port-opening app, in which *NWP* can reach two sensitive behaviors—*s1* and *s2*. In this example, conditional judgment statements numbered 4 and 5 would be excluded from *CCJSs*, because they are not the conditional judgment statements that must be passed through from *NWP* to *s2*. Conditional judgment statements numbered 2 and 3 would also be excluded from *CCJSs*, because more than one jump branch of them can lead to the target sensitive behavior.

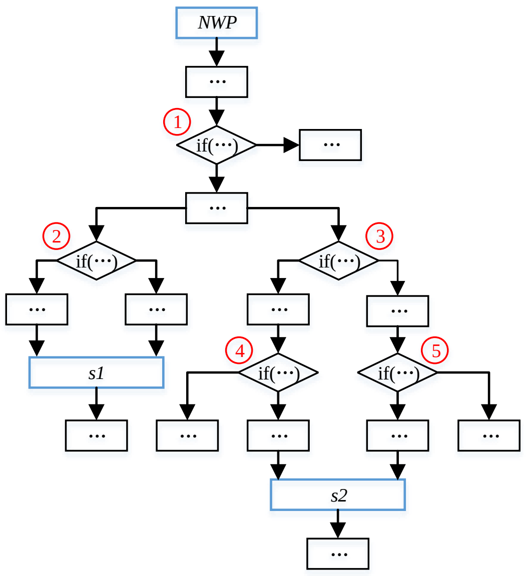


Figure 4. Critical Conditional Judgment Statement Searching

Program Slicing and Strength Judgment. We use *TaintIn* to represent the Java objects that tainted by *NDIB* during the process of taint analysis that started from *NDIP* (i.e., the first kind of taint analysis introduced in Section 3.2). *TaintIn* is maintained and the value of which is changing during the process of taint analysis.

APOVD first performs backward program slicing for each *CCJS c* in C_s. The slicing algorithm of APOVD contains two aspects—intra-procedural and inter-procedural slicing, which is based on the program slicer of Indus[7] and has made some improvements on it. Then the generated program slices are used by APOVD to judge the strength of access control in the path from *NWP* to *s*. The judgment method is that if there exists a *CCJS c* in C_s that satisfies one of the following conditions, APOVD considers that there exists adequate access control in the path from *NWP* to *s*:

(1) The slice corresponding to *c* can lead to GUI changes (e.g., a popup window).

(2) There exists a comparison between two Java objects *obj1* and *obj2* in the slice corresponding to *c*, and $obj1 \in TaintIn$, $obj2 \notin TaintIn$, *obj2* is not composed of constant values and the comparison result can affect the branch jump of *c*.

The first condition is used to judge whether the network request can draw user's attention and get user's consent, which is achieved by searching the slice for the GUI event handler class or method (e.g., GUI event handler class "*OnClickListener*", GUI event handler method "*onClick*", etc.). The second condition is used to judge whether the network request carries a credential that will be validated by the app and the credential is not easy to be cracked (i.e., *obj2* is not composed of constant values). In order to judge whether *obj2* is composed of constant values, APOVD performs backward program

[7] Indus. http://indus.projects.cs.ksu.edu/

slicing for *obj2*, and a constantpropagation analysis would also be done in the generated slice.

In order to improve the performance of strength judgment of access control, APOVD maintains a whitelist and a blacklist to summarize the characteristics of adequate access control and inadequate access control respectively. The whitelist and blacklist are built manually by our analysis work for large amounts of port-opening apps, and they can be easily used to judge the strength of access control by the method of character code matching. For instance, if the random-number generation API invocation "*random*()" can be found in the generated slice for *obj2*, it means that *obj2* is very likely to be dynamic random generated. Then the corresponding *CCJS* is considered as an adequate access control according to APOVD's whitelist matching strategy. If API invocation "*getRemoteAddr()*" or "*getHeader("referer")*" can be found in the generated slice corresponding to *CCJS c*, it means that the access control strategy corresponding to *c* is very likely to be source IP validation or referer validation. Then *c* is considered as the inadequate access control according to APOVD's blacklist matching strategy, because the source IP of a network request and the referer field of an Http request can be easily forged.

4. Evaluation of APOVD

In Section 2, we found 1187 port-opening apps (406 apps in Google Play and 781 apps in Wandoujia) in the 15600 apps downloaded from Google Play and Wandoujia. In this section, we use APOVD to detect port-opening vulnerability in these 1187 apps, followed by the discussion of effectiveness and efficiency of APOVD.

4.1. Detection Result

Sensitive Behavior. APOVD was used to detect these 1187 port-opening apps introduced in Section 2, and the result shows that 375 apps have port-opening vulnerability. We further validated these 375 apps manually, and 329 apps were identified as having port-opening vulnerability, i.e., the false positive rate is 12.27% ((375-329)/375). Besides, we checked the rest 812 (1187-375) apps manually, and the result showed that 78 apps of them have port-opening vulnerability, i.e., the false negative rate is 9.61% (78/812). Therefore, we finally identified that there exists 407 apps (329+78) in these 1187 apps that have port-opening vulnerability. Table 4 shows the 10 most common sensitive behaviors detected by APOVD in these 1187 port-opening apps and the numbers of apps corresponding to them. In the last three columns, the number of apps is further divided according to the strength of access control. Only inadequate or none access control indicates port-opening vulnerability, and the adequate access control does not indicate port-opening vulnerability. It can be seen from Table 4 that, due to the lack of safety consciousness of the developers, large numbers of port-opening apps lack of adequate access control, which may be vulnerable to the attacks from the outside.

Access Control. Based on the program slices generated by APOVD, a statistical analysis work was done to analyze the access control strategies of the 1187 port-opening apps. We list the most commonly used six access control strategies and the corresponding number of apps in Table 5. The most commonly used adequate access control strategies include dynamic credential validation, user authorization and Internet server validation. Internet server validation means that the credential carried by a network request would be sent to an Internet server to be validated. The most commonly used inadequate access control strategies include source IP validation, referer validation and hard coded credential validation. Source IP validation and referer validation can be easily bypassed by forging the source IP of a network request or the referer field of an Http request, therefore both of them are considered as inadequate access control. Hard coded credential validation means that the credential carried by a network request would be validated by the credential that hard coded in the app. As the credential can be easily cracked, therefore it is also considered as inadequate access control.

Table 4. Top 10 Sensitive Behaviors

Sensitive Behavior	App Count	Access Control		
		Adequate	Inadequate	None
Get device Id	284	61	97	126
Get location information	241	62	103	76
Get network information	216	58	77	81
Read file	186	53	78	55
Get app's information	166	47	56	63
Open target component	136	33	49	54
Download file	135	56	37	42
Open web pages	104	45	38	21
Read phone contacts	49	29	12	8
Read SMS records	33	17	9	7

Table 5. Access Control Statistics of Port-opening Apps

Type	Control Method	App Count
Adequate access control	Dynamic credential validation	164
	User authorization	70
	Internet server validation	36
Inadequate access control	Source IP validation	130
	Referer validation	98
	Hard-coded credential validation	29

4.2. Efficiency Analysis

Analysis Time. APOVD's main program ran in a desktop computer with Intel Core i7 CPU, 32G RAM and Windows 7 system. It cost APOVD 89 hours to detect the 1187 apps. The average detection time for each app is 4.5 minutes, which we think is an acceptable time. We found in the experiment that the detection time is approximately proportional to the size of CFG. The size of CFG determines the scope of the static analysis, which makes the detection time be independent of the size of app because APOVD builds the CFG of an app by extracting the code that related to the opened ports, rather than the whole code of an app. We also made a statistical analysis for the time costs of each main process of APOVD, and the result is shown in Table 6. Table 6 lists the average time costs of each main process. It can be seen that the most time-consuming processes are taint analysis and program slicing.

Table 6. Distribution of Analysis Time

Process	Time (s)
CFG Construction	23
Reachability Analysis	31
Taint Analysis	74
Critical Judgment Statement Search	36
Program Slicing	65

Memory Consumption. According to our observation in the experiment, APOVD requires 2GB to 16GB memory to detect an Android app, which can also be considered as an acceptable memory consumption. To build CFG, APOVD needs to construct complex data structures in memory, and the space complexity of these data structures is proportional to the size of code that need to be analyzed in an app. Besides, the processes of taint analysis and program slicing are also very memory-consuming. For taint analysis, it is needed to maintain a large data structure in memory to store all the tainted data during the process of taint analysis. For program slicing, the construction of control dependency and data dependency for an app can lead to large memory consumption.

4.3. Accuracy Analysis

In Section 4.1, we introduced that the false positive and false negative rate of APOVD are 12.27% and 9.61% respectively. In order to analyze the causes of false positives and false negatives, we did a manual analysis for the apps that are missed or false reported by APOVD. The result shows that false negative is mainly caused by the inaccuracy of reachability analysis and taint analysis of APOVD. These inaccurate factors include the start-up relationship between components, reflection mechanism, dynamic loading, native code, etc. For instance, some network requests are sent to another component to be handled, but APOVD cannot guarantee that the target component can be accurately identified by the existing capacity of static analysis. Reflection and dynamic loading mechanism have always been the difficult problems of static analysis, but some of the apps that are missed reported by APOVD apply the reflection or dynamic loading mechanism to handle the network request, therefore, they cannot be accurately detected by APOVD. Besides, as APOVD is mainly used to detect the Java code of Android app and cannot analyze the native code of an app, therefore, if an app applies native code to handle network request, APOVD cannot properly handle it. It is also an important reason that lead to the false negative of APOVD.

False positive is mainly caused by the inaccuracy of program slicing of APOVD. Program slicing is used to judge the strength of access control, and the inaccuracy of program slicing would lead to the wrong judgment of the strength of access control. The accuracy of program slicing depends on the completeness of CFG and the accurate analysis of data dependence and control dependence. However, because of the limitations of static analysis which were mentioned in the previous paragraph, the completeness of CFG can hardly be guaranteed. In addition, the analysis of data dependence and control dependence faces the problems of concurrency, exceptions, heap-allocated data, etc. [17], which have always been the difficulties of Java static program slicing and have not been effectively solved by APOVD. All of them are important reasons that lead to the false positive of APOVD.

4.4. Case Study

Apps open ports for interaction between web page and Android app. Several SDKs produced by Baidu have port-opening vulnerability, and these SDKs are widely used by lots of Android apps. We observed that when visiting some of the web pages of Baidu, the JS codes run in them would send Http requests to some specified network ports to collect private information, and these network ports are precisely the ports that opened by the vulnerable apps that use Baidu SDKs. Table 7 shows the analysis results of port-opening vulnerability of three typical SDKs. It can be seen from Table 7 that these Baidu SDKs do not apply adequate access control and can be easily bypassed to implement attacks. The first SDK (*com.baidu.android.moplus*) is the Baidu wormhole vulnerability which have been introduced in Section 1, and it means that though this vulnerability was published more than one year ago, some apps that use the vulnerable SDK still have not been repaired. The second SDK (*com.baidu.frontia.module*) are used by large amounts of apps (158 apps), and the security problems caused by it are mainly privacy leakage, which can be considered as a not serious security problem. The third SDK (*com.baidu.mobads.remote*) can lead to the problem of file downloading without access control, and it can be considered as a serious security problem.

Some map apps open network ports for receiving and handling Http requests from the corresponding map web sites and providing some necessary information. For instance, Amap (*com.autonavi.minimap*) app opens port 6677, and when the web site of Amap is visited, the JS code in the web page would send Http request to this opened port. By this way, the geographical location responded by Amap app can be shown in the web map of Amap. However, Amap of some versions has port-opening vulnerability, and the vulnerability details are shown in Table 8. The access control Amap used is only referer validation and it can be easily bypassed by forging the referrer field of an Http request. The first two vulnerabilities of Amap shown in Table 8 can lead to privacy leakage, and the third vulnerability can be used to implement fishing attack by luring the user to visit a malicious web page.

Table 7. Statistics of Port-opening Vulnerability of Baidu SDKs

Package Name	Count	Access Control	URL Prefix	Possible Attack
com.baidu. android.moplus	11	Source IP validation	geolocation	Leak location information
			getapn	Leak current network status information
			sendintent	Send any intent to open any web page or interact with other apps
			getcuid	Leak IMEI number
			scandownloadfile	Scan and download files
			addcontactinfo	Add contacts to mobile phones
			getapplist	Leak the information of the installed apps
			downloadfile	Download any file to the specified path and the apk file would be installed automatically
			uploadfile	Upload any file to the specified path and the apk file would be installed automatically
com.baidu. frontia.module	53	Referer validation	geolocation	Leak location information
			getapn	Leak current network status information
			getcuid	Leak IMEI number
com.baidu. mobads.remote	26	None	getfile	Downlaod specified file
			getcuid	Leak IMEI number

Apps open ports for PC Manages Android Device. In the cases of wireless connection between PC and Android device, we found that JoinMe (*com.joinme.maindaemon*) of some versions opens an FTP service by port 2121, but the default username and password of the FTP are hard coded in the app. Therefore the attacker can use the default username and password to visit FTP service of JoinMe. Besides, JoinMe uses the method of verification code authentication to build the connection between the computer and mobile device. Each request from the computer must carry the verification code (SecretKey). However, we found through experiments that there exists leakage of SecretKey in JoinMe. When sending an Http request which carried the command "*JoinMe Broadcast*" to the port 65532 opened by JoinMe, the SecretKey that had been validated would be acted as a part of an Http response. With the obtained SecretKey, the attacker can control the device in the LAN environment and perform each control functions that JoinMe have.

Besides, we also found some vulnerabilities in the cases of wired connection. For instance, Mgyun (*com.mgyun.shua.protector*) would open port 22568 when the computer connects with the mobile device by USB. It keeps on listening on this port even if the USB connection has been broken for several minutes. We also found through experiments that Mgyun does not have access control and it does not validate the legitimacy of any request during the period of opening network port of an app. Therefore, the attacker can construct specified Http requests in the LAN environment to manage the mobile device and perform lots of dangerous operations.

Table 8. Port-opening Vulnerability of Amap

URL Prefix	Possible Attack
geolocation	Leak location information
getpackageinfo?packagename=xxx	Leak the app information of the specified package name
androidamap?action=openFeature& featureName=OpenURL&url=evilsite	Lure the user to visit malicious web page to implement phishing attack

5. Discussion

Prevention Method of Port-Opening Vulnerability. Port-opening vulnerability is a very serious vulnerability of Android app and has been found in many Android apps. To prevent port-opening vulnerability, the developers of Android app should reduce the sensitive behaviors that are exposed to the outside by the opened port as much as possible, and adequate access control is necessary to be used to prevent the possible attacks. The typical adequate access controls include dynamic credential validation, user authorization, Internet server validation, etc., which are introduced in Section 4.1. All of them can be used to effectively prevent port-opening vulnerability. However, perhaps the most fundamental method is try to avoid opening network port outwards. According to our study, many port-opening apps actually do not need to open port to accomplish their necessary tasks. For example, the apps that open port for the communication between computer and Android device can make the programs that run in PC to be acted as the sever side to receive and handle the request from Android app, which can also achieve the necessary communication between PC and Android device, and it had been validated by our experiments.

Limitations of AOPVD. AOPVD can be used by the Android app market or ordinary users to detect the port-opening vulnerability of Android app, and reduce the potential security threats brought by the vulnerability. Though AOPVD is effective in detecting port-opening vulnerability of Android app, it still has some limitations that need to be improved:

(1) AOPVD cannot handle the native code. AOPVD can only be used to detect the port-opening vulnerability that exists in the Java code of an app. However, according to our study, a small number of Android apps implement the opening network port by the native code. In this condition, AOPVD cannot accurately detect the port-opening vulnerability in these apps.

(2) The completeness of CFG can hardly be guaranteed. The static analysis methods of AOPVD are mainly based on CFG, but the construction of CFG faces the challenge of implicit paths in Android, such as the implicit paths that achieved by callbacks, intents and other Android-specific facilities. This challenge is not limited to AOPVD, but is a

common problem for the static analysis of Android app. Besides, some mechanisms in Java can also influence the construction of CFG, e.g., reflection, dynamic loading, JNI (Java Native Interface), etc., and they have always been a difficult problem for Java static analysis.

(3) Accuracy of program slicing. Program slicing of AOPVD faces the problems of concurrency, exceptions, heap-allocated data, etc., which have always been the difficulties of Java static program slicing and have not been effectively solved by AOPVD. Therefore, how to improve the accuracy of AOPVD's program slicing is a difficult problem that we must solve in the future work.

Future work. In the future work, we will take the native code into consideration and propose the detection method that can be used to detect the port-opening vulnerability in native code. Effective measures will be found and taken to overcome the difficulties AOPVD faces in construction of CFG. We will also improve our program slicing algorithm to achieve a more accurate judgment of access control and smaller rate of false positives. Finally, we will recommend AOPVD to the ordinary users or owners of Android app stores to detect the port-opening vulnerability of the apps in their devices or stores to reduce the possible attacks that caused by the vulnerability.

6. Related Work

Because port-opening vulnerability is a new vulnerability type in Android application layer, therefore we first review the typical vulnerabilities found by the researchers in Android application layer, along with some discussions about the possible attacks.

Among the numerous vulnerabilities in Android application layer, vulnerabilities caused by exported component and vulnerabilities caused by WebView drew most of the attention of the researchers [2]. A vulnerable and exported component can be visited by the other apps in the same device, which can lead to many types of vulnerabilities, e.g., capability leak [1,2], content provider leakage [3,4], privilege escalation [5,6], confused deputy [7], component hijacking [8,9], intent spoofing [10], etc. But these vulnerabilities can only be exploited locally (the vulnerable app and the malicious app must run in a same device). They cannot be exploited remotely in the LAN environment like port-opening vulnerability.

WebView vulnerability is another important vulnerability type in Android application layer that attract many researchers' attention [12]. Numerous attack forms have been found and studied by the researchers, such as excess authorization [11], file-based cross-zone scripting [18], touchjacking [19], unauthorized origin crossing [20], etc. WebView vulnerabilities are mainly caused by the insecure API invocations from the JS code to the Java or native code in an app. To successfully exploit these vulnerabilities, the attackers must first take measures to make the malicious JS code to run in the vulnerable app by WebView. These measures include alluring the victim to open a web page which contains malicious JS code or intercepting and tampering with Http response in a manner of Man-in-the-Middle (MITM) attacks. However, compared with the exploit methods of port-opening vulnerabilities, these exploit methods are more difficult to be implemented.

Perhaps the most relevant work to ours is Wang's work [21]. Wang et al design and implement OPAnalyzer which can effectively identify and characterize vulnerable open port usage in Android applications. But OPAnalyzer can only be used to analyze program behavior of port-opening Android apps, rathar than automatically detect port-opening vulnerability like APOVD.

7. Conclusion

In this paper, we propose a new type of vulnerability—port-opening vulnerability of Android app, which exists widely in the current Android apps. We also propose the detection method of port-opening vulnerability based on static analysis and implement a detection tool—APOVD. 1187 port-opening Android apps are found in 15600 popular apps downloaded from Google Play and Wandoujia, and 407 of them are identified as having port-opening vulnerability with the help of APOVD. The result shows that APOVD is effective in detecting port-opening vulnerability.

References

[1] P. P. Chan, L. C. Hui, and S.-M. Yiu, "Droidchecker: Analyzing Android Applications for Capability Leak", Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Tucson, AZ, USA, **(2012)** April 16-18.

[2] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones", Proceedings of the 19th Annual Network & Distributed System Security Symposium, San Diego, California, USA, **(2012)** February 5-8.

[3] Y. J. Zhou and X. X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications", Proceedings of the 20th Network and Distributed System Security Symposium, San Diego, California, USA, **(2013)** February 24-27.

[4] H. Shahriar and H. M. Haddad, "Content Provider Leakage Vulnerability Detection in Android Applications", Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, UK, **(2014)** September 9-11.

[5] P. P. Chan, L. C. Hui, and S. Yiu, "A Privilege Escalation Vulnerability Checking System for Android Applications", Proceedings of IEEE 13th International Conference on Communication Technology, Jinan, China, **(2011)** September 25-28.

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android", Proceedings of the 13th International Conference on Information Security, Boca Raton, FL, USA, **(2010)** October 25-28.

[7] J. Wu, T. Cui, T. Ban, S. Guo, and L. Cui, "Paddyfrog: Systematically Detecting Confused Deputy Vulnerability in Android Applications", Security and Communication Networks, vol. 8, no. 13, **(2015)**, pp. 2338-2349.

[8] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities", Proceedings of the 19th ACM Conference on Computer and Communications Security, Raleigh, NC, USA, **(2012)** October 16-18.

[9] M. Zhang and H. Yin, "Appsealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications", Proceedings of the 21th Annual Network & Distributed System Security Symposium, San Diego, California, USA, **(2014)** February 23-26.

[10] D. Kantola, E. Chin, W. He, and D. Wagner, "Wagner. Reducing Attack Surfaces for Intra-Application Communication in Android", Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Raleigh, NC, USA, **(2012)** October 16-18.

[11] E. Chin and D. Wagner, "Bifocals: Analyzing Webview Vulnerabilities in Android Applications", Proceedings of the 14th International Workshop on Information Security Applications, Jeju Island, Korea, **(2013)** August 19-21.

[12] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on Webview in the Android System", Proceedings of the 27th Annual Computer Security Applications Conference, Orlando, FL, USA, **(2011)** December 5-9.

[13] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically Detecting Implicit Control Flow Transitions Through the Android Framework", Proceedings of the 22th Annual Network & Distributed System Security Symposium, San Diego, California, USA, **(2014)** February 8-11.

[14] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the Android Permission Specification", Proceedings of the 19th ACM Conference on Computer and Communications Security, Raleigh, NC, USA, **(2012)** October 16-18.

[15] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-Learning Approach for Classifying and Categorizing Android Sources and Sinks", Proceedings of the 21th Annual Network & Distributed System Security Symposium, San Diego, California, USA, **(2014)** February 23-26.

[16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps", ACM SIGPLAN Notices, vol. 49, no. 6, **(2014)**, pp. 259-269.

[17] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer, "A New Foundation for Control Dependence and Slicing for Modern Program Structures", ACM Transactions on Programming Languages and Systems, vol. 29, no. 5, **(2007)**, pp. 27.

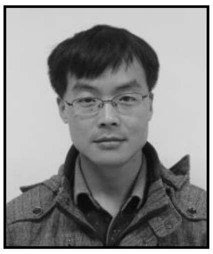
[18] D. Wu and R. K. Chang, "Analyzing Android Browser Apps for File://Vulnerabilities", Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, UK, **(2014)** September 9-11.

[19] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, “Touchjacking Attacks on Web in Android, IOS, and Windows Phone”, Proceedings of the 5th International Symposium on Foundations and Practice of Security, Montreal, QC, Canada, **(2012)** October 25-26.

[20] R. Wang, L. Xing, X. Wang, and S. Chen, “Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation”, Proceedings of the 20th ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, **(2013)** November 4-8.

[21] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao, “Open Doors for Bob and Mallory: Open Port Usage in Android Apps and Security Implications”, Proceedings of the 2nd IEEE European Symposium on Security and Privacy, Paris, France, **(2017)** April 26-28.

Authors



Hongzhou Yue is currently a Ph.D. candidate in Department of Communication Engineering, Xidian University, China. He has joined in State Key Laboratory of Integrated Services Networks in Xidian University since 2013. His research interests include Android security and web security.

Yuqing Zhang is a professor in National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing, China. He received his B.S. and M.S. degree in Computer Science from Xidian University, China, in 1987 and 1990 respectively. He received his Ph.D. degree in Cryptography from Xidian University, in 2000. He is a member of IEEE Communications Society and IEICE Transactions on Communications. His research interests include computer network, cryptography and network protocol security.