Static Analysis of Superfluous Network Transmissions in Android Applications

Jianmeng Huang, Wenchao Huang, Zhaoyi Meng, Fuyou Miao, and Yan Xiong
(Corresponding author: Wenchao Huang)

School of Computer Science and Technology, University of Science and Technology of China
Elec-3 (Diansan) Building, West Campus of USTC, Huang Shan Road, Hefei, Anhui Province, China
(Email: huangwc@ustc.edu.cn)

(Received June 20, 2018; Revised and Accepted Nov. 22, 2018; First Online July 16, 2019)

Abstract

The network transmission is an important way to exchange information between Android applications and their own backend or other third-party servers. However, some network transmissions are superfluous for the apps' functionalities. Superfluous network transmissions not only increase the network traffic but also may leak users' sensitive data. To identify the superfluous network transmissions, we propose a static-analysis based approach. Evaluation with real world market apps shows that 62% apps contain superfluous network transmissions, and 48% of the analyzed network transmissions are superfluous, and our approach could effectively detect superfluous network transmissions in Android apps.

Keywords: Android Security; Privacy Leakage; Superfluous Network Transmissions

1 Introduction

In recent years, smartphone plays an important role in people's daily life. It is not simply a communication tool now, but also a data container and a personal assistant. Various functionalities of smartphones are provided by multifarious applications (apps), which can be downloaded from the app market or third parties. In 2017, the number of available apps in the Google Play Store was placed at 3 million apps [19]. As the development of Android apps, the functionalities provided by apps become more refined and personal customized. As a result, sensitive data are collected and may be transmitted via network to support these functionalities [20]. For example, an app which sells movie tickets may utilize the GPS data of the smartphone, transmitting the GPS data to remote servers and getting the recommendation of nearby cinemas.

While some network transmissions are needed to fulfill apps' functionalities, other network transmissions are superfluous. The superfluous network transmission means that the transmission is not necessary: No matter the transmission is success or not, it is of no help for the apps functionalities. Some malicious apps may intentionally leak users' privacy via network transmissions [8]. These network transmissions are superfluous and have no aid to the app functionalities. Even in benign apps, there may also be superfluous network transmissions which collect users' privacy. The superfluous network transmission does not benefit users. First, the superfluous network transmission increases the network traffic and consumes the power resource of mobile devices. Second, the superfluous network transmission may leak users' privacy. Since the transmission is not necessary for app functionalities, it is of high possibility that the transmission is useful to app providers. For example, an app provider may collect users data for advertisement purpose or user habit analysis.

Existing techniques are insufficient in detecting such superfluous network transmissions. Rubin *et al.* propose a technique [16] which focuses on detecting covert communications that have no effect on the user-observable application functionality. Its core idea is to look for cases when no information is presented to the user neither on success nor on failure of the connection. We argue that this definition about covert communication is not proper. First, covert communication could also present information to the user when the connection failure occurs, *e.g.*, when a device is put in disconnected environment or airplane mode, for that warning the user about network failure would not expose the purpose of the malicious network communication. Second, some network transmissions could also be necessary for app functionalities even though these transmissions have no direct effect on the user interface. For example, at initialization, an app may synchronize data from remote server and store it, which is not used by the app immediately but used latter by other app functionalities. Hence, it is not always appropriate to distinguish necessary network transmissions and superfluous ones by figuring out whether the transmissions (either on success or on failure of the transmission) could directly result in affecting the user interface. LeakSemantic [10] targets for locating abnormal sensitive network transmissions from mobile apps. It consists of a program analysis component to precisely identify sensitive transmissions and a machine learning component to further differentiate between the legal and illegal network transmissions. It focuses on the sensitive data, but users' behavior data are also privacy. Besides, it uses lexical features derived from the set of URLs in the traffic traces to train classifiers, which only works for connections using `HTTP GET` request.

In this paper, we propose a novel approach to detect the superfluous network transmissions in Android apps. We model superfluous network transmissions as the transmissions of which the responses are not utilized by apps. Our key insight is that if a network transmission is necessary, the response of the network connection should be utilized by the app. Our approach decides whether a network transmission is necessary by the information of how the response of the network transmission is handled. It is different from existing researches [10, 23, 25] that detect network transmissions by figuring out how sensitive data are generated, utilized and finally transmitted out of the device. Our approach concentrates on figuring out whether the responses of network transmissions are utilized by the app. If not, we consider corresponding network transmissions as superfluous. Our approach is also different from the research [16] which detects covert communications that have no effect on the UI. We argue that only if the response of the network communication is not utilized by the app, the communications is superfluous. Overall, our approach detects superfluous network communications from a different aspect. Existing researches could be complementary to our work and they can work with our approach side by side to enhance user privacy.

To figure out whether the response is used by the app, we utilize the information flow analysis to track how the response is used. Here, we address two challenges. First, how to choose the `sources` and `sinks` of the information flow analysis. Improper `sources` and `sinks` may lead to the insufficient identification or false positives. Second, we use a novel light-weighted approach to handle the implicit data flow of the response.

Our contributions are summarized as follows.

- We study superfluous network communications and propose a new way of distinguishing superfluous and necessary network communications.
- We propose a static analysis approach which automatically detects superfluous network transmissions in Android apps.
- We demonstrate the effectiveness of our approach with real-world Android applications.

The rest of this paper is organized as follows. Section 2 introduces the background and states the problem. Section 3 details the design of our approach and after that, Section 4 describes experimental results. Section 5 discusses the future work of our approach and Section 6 describes related work. Finally, Section 7 concludes the paper.

2 Background and Problem Statement

2.1 Network Transmissions in Android Apps

In order to perform network operations in Android applications, many APIs are developed. Table 1 lists the base classes and methods which are responsible for network connections and data transmissions. The third column of the table lists the methods which are responsible for getting the return value of network connections. Most network-connected Android apps use HTTP to send and receive data. Besides, developers could also use other basic JAVA network connecting APIs, which is listed in the last four rows in the table.

Developers should use an asynchronous task for network transmissions so the UI thread doesn't freeze. If the UI thread freezes, Android will show an "Application not responding" dialog to the user. To avoid creating an unresponsive UI, it is recommended not to perform network operations on the UI thread. By default, Android 3.0 and higher versions require apps to perform network operations on a thread other than the main UI thread; if not, a `NetworkOnMainThreadException` is thrown. To facilitate the deployment of network transmission, many third-party libraries provide APIs to encapsulate asynchronous network operations. Generally, these libraries are implemented based on the base classes and methods.

In some network transmission libraries, using asynchronous requests forces developers to implement a `Callback` with its two callback methods: success and failure (*i.e.*, `onResponse` and `onFailure()`). When calling the asynchronous `getTasks()` method from a service class, developers have to implement a new `Callback` and define what should be done once the request finishes.

2.2 Problem Statement

```
public String sendData(){
  String message;
  try {
    OkHttpClient client = new OkHttpClient
        ();
    FormBody.Builder formBody = new
        FormBody.Builder();
    formBody.add(``username'',``foo'');
    Request request = new Request.Builder
        ()
      .url("http://www.sample.com")
      .post(formBody.build())
      .build();
    Response response = client.newCall(
        request).execute();
 // if (response.isSuccessful()) {
 //    message = response.message();
 // }
  } catch (Exception e) {
    e.printStackTrace();
  }
```

Table 1: The considered network connection APIs

Class or Interface	**Connecting**	**Getting Response**
java.net.URLConnection	Connect	GetInputStream
java.net.URL	OpenConnection	OpenStream
org.apache.http. client.HttpClient	Execute	Execute
java.net.Socket	GetInputStream getOutputStream	GetInputStream
com.squareup. okhttp.OkHttpClient	NewCall	NewCall

```
18   return message;
19 }
```

Listing 1: An example of network transmission

As a motivating example, Listing 1 shows an example of superfluous network transmission. The network transmission is implemented based on the `okhttp`. It sends the keyword *username* to the remote server by HTTP POST request. If the response of the network connection is not used by the app (*i.e.*, the codes commented from line 12 to line 14), the function of method `sendData` is simply sending data to the remote server. This network transmission may be useful for the app provider, but it is of no use for the app's functionalities for that the app does not gain any feedback from the network transmission.

To handle the problem, we face the challenge of deciding whether the response of a network transmission is utilized by the app. First, we should trace how the response is used and figure out whether it is used for necessary app functionalities. Second, there may be implicit information flow using the response, which increases the difficulty of tracing the use of the response.

3 Design and Implement

We propose a static analysis approach to find the superfluous network transmissions in Android applications. This section describes how we model the superfluous network transmissions and how we identify them in Android apps.

3.1 Overview

The core idea behind our approach is to look for cases where the responses of network transmissions are not utilized by apps. We determine whether a network transmission is superfluous by tracking how the response is used by the app. If the response is not utilized by the app or improperly used, we deem the network transmission superfluous.

Guided by this idea, Figure 1 shows the work-flow of our approach. Given an app, we first translate the apk file of it into an intermediate representation (*e.g.*, Jimple representation, a statement based intermediate representation), based on which we could apply static analysis. This process is commonly used by static analysis for Android apps [1, 4, 22]. Then we search the code segments which are responsible for network transmissions in the

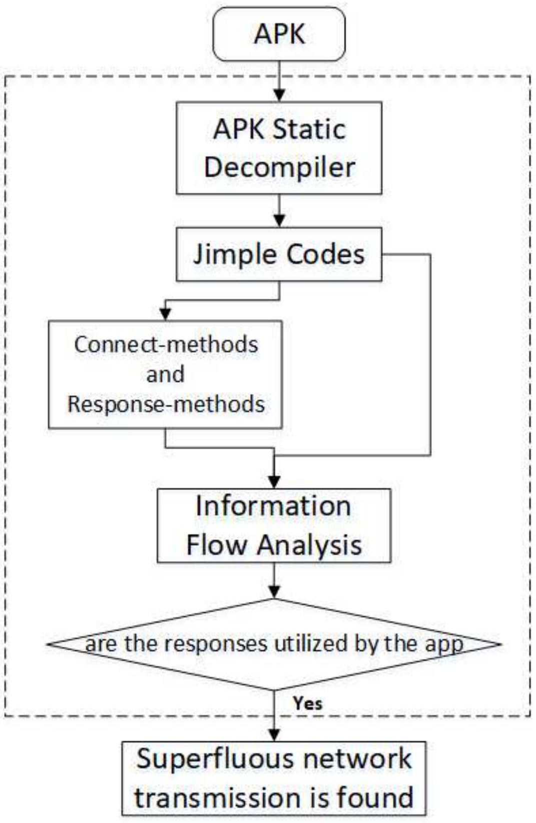

APK
APK Static Decompiler
Jimple Codes
Connect-methods and Response-methods
Information Flow Analysis
are the responses utilized by the app
Yes
Superfluous network transmission is found

Figure 1: Work-flow of our approach

intermediate representation. Afterwards, we apply information flow analysis to check whether the responses of network transmissions are utilized by app functionalities. Here, the information flow analysis tracks how sensitive information is propagated through an application. Finally, we find the superfluous network transmissions by identifying unused responses.

To apply the information flow analysis, we use FlowDroid [1] as the underlying analysis infrastructure. FlowDroid provides taint analysis which presents potentially data flows to human analysts or to automated app-detection tools which can then decide whether a data use actually constitutes a policy violation. It adequately models Android-specific challenges like the application lifecycle or callback methods, which helps reduce missed leaks or false positives. Our analysis is based on information flow analysis of FlowDroid on Android apps. Information flow analysis is a common technique used for analyzing Android apps in that it can track how data are potentially be used by the app, which helps to distinguish different use of the data (*e.g.*, improper use and legitimate use).

We use static analysis based approach because it is usually more efficient [13]. Static analysis is applied without executing the code. It relies on Java bytecode extracted by disassembling an application. As a result, static analysis could help analyst to inspect all the behaviors of an app. We do not adopt dynamic analysis approach because dynamic analysis faces the problem of low testing coverage, which causes insufficient analysis of the app. In dynamic analysis, analysts have to execute (or emulate) the app in order to collect runtime information used for further analysis. However, it cannot be guaranteed that all the code paths in an app are executed during the dynamic analysis, because some code paths require particular trigger conditions. For example, for a shopping app, an app behavior may only occur when the user buys something. Besides, dynamic analysis executes only one code path of an app at one time, while static analysis could be applied in parallel. Overall, compared with static analysis, it is not efficient to test apps with dynamic analysis.

3.2 Design

In this subsection, we describe how we find the superfluous network transmissions. We first introduce cases which are considered to be superfluous network transmissions. Then we describe how our static analysis approach handles these cases.

3.2.1 Models of Superfluous Network Transmissions

To automatically detect superfluous network transmissions in Android apps, we first study network transmissions in malicious and benign Android apps to find the superfluous ones and draw the common points of them. We also study the different features between legitimate network transmissions and superfluous ones. Particularly, we denote the *connect method* as method which is responsible for creating connections between devices and remote servers, and we denote *response method* as method which is responsible for getting responses from the connection. Table 1 shows the *connect methods* and *response methods* monitored by our approach. Overall, we summarize the following three categories as superfluous network transmissions considered in this paper.

Category 1: A network transmission calls the *connect method* but does not call the *response method*. In this category, the network connection is established, but the response of the connection is not handled. As a result, the transmission simply sends data to remote server. The app is not going to establish the connection again if the transmission fails. Usually, for a necessary network transmission, the app would at least query the status of the transmission (*e.g.*, querying the HTTP status code) to check if the network transmission fails. If it fails, the app would handle the failure in an exception and then inform users that the network is not available. Network transmissions in this category are deemed superfluous for that they are not properly deployed and the purposes of these transmissions are not clear. These superfluous network transmissions may be caused by careless app developing or malicious privacy collecting which avoids being noticed by users.

Category 2: A network transmission calls both *connect method* and *response method*, but the app does not actually utilize the response of the network transmission. At the code level, the *response method* is invoked, but the value of the response is not passed to other codes (*i.e.*, `methods` developed by the app) in the app. In this category, although the app gets the response of the network transmission, it does not make use of the response for the app functionalities. Hence, the network transmission does not contribute to the normal app functionalities. Network transmissions in this category may be caused by the iterative development of the app. Some functionalities of the app are discarded but the corresponding codes are not clearly removed.

Category 3: A network transmission calls *connect method* more than one time, and the URLs of the connection are different. We find that some apps send the same data to multi servers simultaneously. One of the transmission addresses belongs to the app provider, but other ones are data centers. We consider that at least one of the network transmissions is superfluous, because the transmitted data are the same and one response from the remote server is enough for the app functionalities. The purpose of network transmissions in this category is that one network transmission is used for necessary app functionalities, and others are used for transmitting users' data to data centers, which consumes the network resources of mobile devices. Note that if the network transmissions are in different branch statements, we do not consider them as superfluous ones. Because the app provider may own multi servers, some of which are alternate servers. Hence, in case the main server is down, the network transmissions in different branches could be established.

3.2.2 Finding Superfluous Network Transmissions

We further develop a static analysis approach to identify superfluous network transmissions in the above categories. To handle the three categories of superfluous transmissions, we develop an algorithm based on information flow analysis. Information flow analysis tracks sensitive "tainted" information through the app by starting at a pre-defined source (*e.g.*, an API method returning the response of a network transmission) and then following the data flow until it reaches a given sink (*e.g.* a method

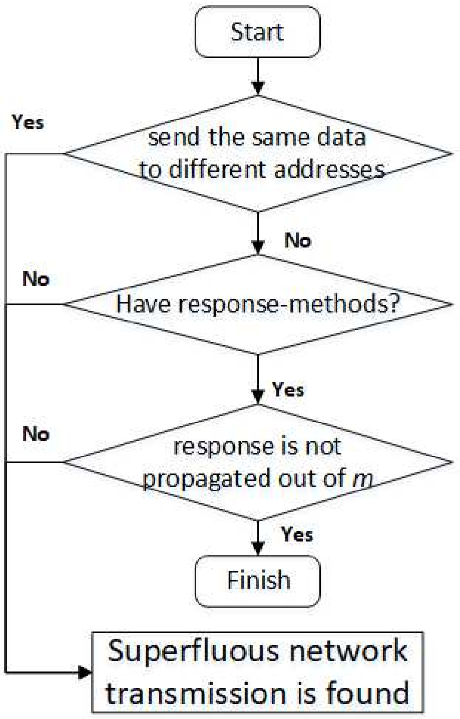
Algorithm 1 Identifying superfluous network transmissions

```
1: Input: The map of connect method and response method $methodMap$, and a method $m$ in the app
2: Output: Whether the app has superfluous network transmissions
3: Begin
4: Count ⇐ times of $methodMap$.key() invoked by $m$
5: if $count > 1$ then
6:   if sending same data to different addresses then
7:     return true
8:   end if
9: end if
10: if $count > 0$ then
11:   for all $connect\_method$ invoked by $m$ do
12:     $response\_method$ ⇐ methodMap[connect_method]
13:     if $response\_method$ is also invoked then
14:       $response$ ⇐ the result of $methodMap[connect\_method]$
15:       if $response$ is not propagated out of $m$ or the callback method then
16:         return true
17:       end if
18:     else
19:       return true
20:     end if
21:   end for
22: end if
23: return false
24: End
```

writing the information to a UI element), giving precise information about which data may be leaked.

Algorithm 1 shows how we identify superfluous network transmissions. It accepts the maps of *connect method* and *response method*, and a method m in the app. Here, the maps of *connect method* and *response method* are pairs of methods of the same network transmission library, and the overloaded methods are also included in the maps. The output of this algorithm is a judgment about whether the app contains superfluous network transmissions. We first decompile the apk file into bytecodes, then we utilize Soot [21] to translate the bytecodes to intermediate representation (*i.e.*, Jimple). Based on the Jimple representation, we get all the `methods` defined by the `classes` in the app. Then, we look for superfluous network transmissions in each method using Algorithm 1.

For each method m in the app, we look for network transmissions and then figure out whether the transmissions are superfluous. Figure 2 illustrates the methodology of Algorithm 1. It first checks superfluous network transmissions in category 3. It figures how many times the *connect methods* are invoked in the method m. If the *connect methods* are invoked more than one times, which is the situation in category 3, we check whether there exist more than two network transmissions send the same data to different addresses. If found, we report the superfluous network transmissions (*i.e.* line 6 to 8).


Start
send the same data to different addresses
Yes
No
Have response-methods?
No
Yes
response is not propagated out of m
No
Yes
Finish
Superfluous network transmission is found
Figure 2: Methodology of identifying superfluous network transmissions

Then, Algorithm 1 checks superfluous network transmissions in category 1, as shown by Figure 2. If *connect methods* are invoked, Algorithm 1 would check each of the *connect methods* (*i.e.*, line 11). Then for each *connect method*, we check whether the corresponding *response method* is called in the method m or whether there is a corresponding callback method (*e.g.*, `onResponse`), which is designed for asynchronous purpose. If not, we report the connection as superfluous one.

Otherwise, Algorithm 1 checks whether the result of the *response method* is utilized by the app (*i.e.*, line 13 to line 17). The process is illustrated by Figure 2. Here, the information flow analysis is used to handle the situation in category 2. Specially, we set the results of the *response methods* as *sources* and the `return` statement of the method as the *sink*. Besides the *response methods*, we also include other APIs, which utilizes the response of network connection, in the *sources*. *e.g.*, HttpURLConnection.getResponseCode(), HttpResponse.getStatusLine(), HttpResponse.getEntity(). As a result, if the response is propagated via the data flow out of the method m or the callback method, which indicates that the response would be utilized by other methods, this network transmission is deemed necessary. Otherwise, the connection is deemed superfluous.

We set the *sources* and *sinks* in two configuration files, which enables the scalability of our approach. As a result, if we find new third party libraries which are responsible for network transmissions, we can add the corresponding methods into our configuration files.

Additionally, we handle the implicit data flow of utilizing responses of network transmissions. Listing 2 shows an example of returning the status of a network transmission. The `response` is used to get the *status code* of the network transmission (*i.e.*, line 6). The `AllSuccess` indicates that whether the connection succeed, but there is no explicit data flow from the `response` to the `AllSuccess`. To handle this situation, we could take advantage of existing approaches, such as EdgeMiner [4], which addresses implicit flows in static analysis. As we only need to handle implicit data flow for specific resources (*e.g.*, `getStatusCode()`), we use a lightweight way to handle this problem.

```
 1 public boolean sendData(){
 2   boolean allSuccess = true;
 3   try {
 4     ...
 5     HttpResponse response = client.execute
           (http, httpContext);
 6     if (response.getStatusLine().
           getStatusCode() != 204) {
 7       allSuccess = false;
 8     }
 9   } catch (Exception e) {
10     e.printStackTrace();
11   }
12   return allSuccess;
13 }
```

Listing 2: Implicit data flow of utilizing response

Our lightweight way specifically monitors the branch statements to address the implicit data flow of using `response`. In detail, we first locate the methods which contain network transmissions in the app. Then we locate the branch statements in these methods. If the conditional statement of the branch calls the APIs which utilize the response of the network connection, we would trace the variables in the branches. Finally, if none of these variable contributes to the variable which is to be returned by the method, we consider this network transmission as superfluous.

4 Evaluation

In this section, we intend to evaluate our approach in the following aspects. First, how effective is our approach in identifying superfluous network transmissions? Second, how often does superfluous network transmission occur in real-world applications?

4.1 Real World Apps Study

We first apply our approach to real-world apps in order to assess its effectiveness. We analyze 12 apps with our approach and manually check the results. As there are no researches or reports about superfluous network transmissions in Android apps, we need to manually inspect the source codes to get the ground truth of the tested apps. Hence, we choose four open-source apps from F-Droid [7], an installable catalogue of free and open source applications for the Android platform. To include commercial apps, other four apps are from wandoujia, a popular third party Android app market, and Google Play. The experiments are conducted on a 4-processor 16GB-RAM machine.

The names and package names of apps, as well as the analysis results, are listed in Table 2. Given the ground truth information (*i.e.*, the results from manual inspecting) and the analysis results, there are four possible outcomes: True positive (TP), true negative (TN), false positive (FP) and false negative (FN). TP means that an app contains superfluous network transmissions with respect to ground truth and our approach detects the superfluous transmissions. TN means that an app does not contain superfluous network transmissions with respect to ground truth and our approach does not find superfluous network transmissions in the app. FP and FN have similar meanings. The metric accuracy is computed by the following formulas:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

For open source apps, our approach finds all the network transmissions, and it detects no superfluous network transmissions in these apps. The results are manually checked by reviewing the source codes. As these apps are open-source, the functionalities of these apps are explicit. If there exist superfluous network transmissions which may leak users' privacy in these apps, the developers would be blamed. Thus there are no superfluous network transmissions which have no use for the app functionalities. As listed in the last column in Table 2, we do not find superfluous network transmissions in these apps both with our approach or manually checking the source codes. The analysis results on open source apps show that our approach has no false positives. Besides, the analysis results have no false negatives. As a result, we get 100% accuracy of analyzing open source apps.

For the market apps, after our automatically analysis, we also manually check the results in the decompiled codes. As these commercial apps does not provide the source codes of them, we utilize dex2jar and jd-gui to get the decompiled Java codes and retrieval the detected results in the decompiled codes. The first 4 apps are from wandoujia market and the last 4 apps are downloaded from Google Play. The results, listed in Table 2, show that our approach successfully finds all the network transmissions in the app and precisely identifies superfluous network transmissions of the three categories. The accuracy of the analysis result is also 100%. Overall, experimental results of the 8 apps demonstrate the accuracy of our approach on commercial apps.

Our experiments also show that the time overhead of our static analysis is acceptable, regarding to the size of each app. Similar to approaches based on information flow analysis [1, 11], it is time-consuming to analyze the app when the size of the app is large, because the method

Table 2: Results of app study (M: manually checked; A: automatically detected)

App sources	App Name	Size	Time	Network transmissions (M/A)	Superfluous (M/A)
Open source	Battery Dog	22K	3s	0/0	0/0
	ArchWiki Viewer	1.1M	6s	1/1	0/0
	Commons	20M	138s	3/3	0/0
	External IP	9.9K	3s	1/1	0/0
App market	zuimei weather	18M	514s	7/7	4/4
	sogou novel	12M	220s	18/18	11/11
	Kugou Music	47M	1672s	10/10	4/4
	karaoke	42M	941s	18/18	6/6
	Vault-Hide SMS	11M	244s	12/12	4/4
	Duolingo	21M	518s	8/8	2/2
	ibis Paint X	31M	729s	9/9	2/2
	Magzter	16M	331s	31/31	12/12

invocation relations in the app become complicated. In our experiments, we observe that the open source apps have smaller sizes than market apps, and the time overhead is lower. The reason may be that most market apps are obfuscated to avoid code plagiarism and vulnerability searching [3]. The released apps are more complexed after obfuscation, increasing the analysis time overhead. Unlike the approach [16] which identifies a network transmission by judging whether the result of the transmission has direct effect on the user interface, our approach adopts some simplifications to improve scalability, such as judging whether the response of a network transmission is utilized by the app by tracing the data flow of the response until the `return` statement of the current method. As a result , we demonstrate that the our static analysis achieves relatively high performance.

Furthermore, we analyze 100 apps downloaded from wandoujia and 100 apps from Google play to figure out how common is the superfluous network transmissions in real world apps. The analyzed apps are the most popular apps in the market collected from different categories such as games, tools, entertainment, weather, social and sports. Experiments on these apps show that for the apps from wandoujia market, 62% of the apps contain superfluous network transmissions. Besides, of all the network transmissions in these apps, 48% of them are identified as superfluous network transmissions by our app. For the apps from Google Play, 22% of the apps contain superfluous network transmissions, and 43% of the network transmissions are identified as superfluous ones. Overall, we can conclude that superfluous network transmissions exist in real world apps with high proportion.

4.2 Finding and Case Study

We find that most of the superfluous network transmissions are collecting users' data to remote servers. Most of the transmitted data are related to user's identity (*e.g.*, device id, product id, IMEI, *etc.*). Besides, the superfluous network transmissions are often triggered by UI elements which are frequently triggered. We can conclude that collecting user's personal data are an important purpose of superfluous network transmissions. For example, *KugouMusic* is a popular music app in the app market. Our approach finds that it has 4 superfluous network transmissions. We decompile the apk file and locate the code segments of the superfluous network transmission. Then we trace back to the event which leads to the superfluous network transmission: as shown in Figure 3, the *Watch* button would lead to the network transmission. In users' expectation, the *Watch* button should only provide the function of switching between different UIs. It should not lead to any network transmission. Hence, this network transmission here is superfluous for the app functionality. Furthermore, we inspect the `address` of the superfluous network transmission at runtime, we find the address belongs an Internet Data Center (IDC) provider in Beijing rather than the command and control server. Hence, we believe that our insight of detecting superfluous network transmission is reasonable and useful.

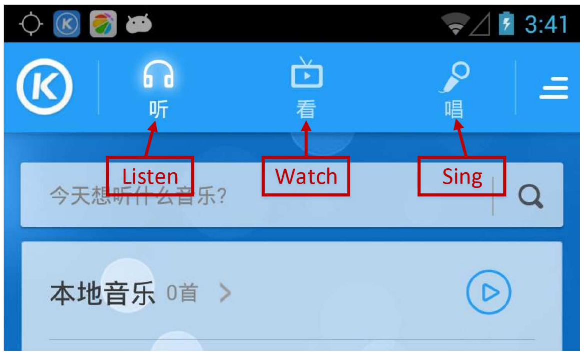


Figure 3: The GUI of KugouMusic

We also find that blocking the superfluous network transmission would not impact the app functionalities. We manually disable the detected superfluous network transmission in the tested apps and repackage the apps. Then we compare the app functionalities between the original apps and repackaged apps. For the two versions of each app, we feed them with the same inputs. If the user interfaces which represent the functionalities are different during the test, *e.g.*, the repackaged app crashes or some information in the user interfaces of the repackaged app is missing, the corresponding app functionality are impacted. Finally, we observe that the blocked superfluous network transmissions have no impact on the normal app functionalities.

5 Discussion

After identifying the superfluous network transmissions in apps, the results could be reported to app markets or app providers. Our approach can be used by app markets to display the detection result of each app, which urges the app provider to make a clear statement about how the app would use users' private data in the privacy policy (listed in the app market). The app could also prompt a privacy collecting request to users. If users do not agree the app to collect their private data, the app should not transmit private data to remote servers. Besides, our approach can be adopted by app providers to check whether the identified superfluous network transmissions are caused by careless coding, which helps to improve the code developing of the app.

The superfluous network transmissions can be blocked to reduce the risk of privacy leakage if the app provider does not provide revised version of the app. There are two possible solutions to automatically block the detected superfluous network transmissions. The first one is to statically disable the code segments which are responsible for superfluous network transmissions. Similar to prior researches [5, 12, 17], we could reduce the unwanted code segments which are responsible for superfluous network transmissions, and then repackage the app. As a result, the repackaged app does not contain superfluous network transmissions. The second solution is to record the patterns of the superfluous network transmissions, and then disable the transmissions at runtime, which could take advantage of the framework of a prior approach [14]. Blocking the superfluous network transmissions is beyond the research scope of this paper and we leave it as our future work.

6 Related Work

There are several approaches to analyze the behaviors of Android apps. FlowDroid [1] and DroidSafe [11] provide static taint-analysis tools to detect potentially malicious data flow in Android applications. Our approach utilizes the data flow analysis provided by FlowDroid. TaintDroid [6] and TaintART [18] propose system-wide information flow tracking tool that can simultaneously track multiple sources of sensitive data. They are dynamic-based approaches and hence face the problem of low test coverage. Amandroid [22] presents a general static analysis framework for security analysis of Android applications. It can precisely track the control and data flow of an app across multiple components, and can compute an abstraction of the app's behavior in the forms of an inter-component data-flow graph and data dependence graph. However, high privacy requires more time and computing resources. Qian *et al.* [15] combine static and dynamic techniques to find potential risks in an app and then embed monitoring code in the app. As a result, their approach could report the content of data transmissions when users are running the app. However, it is not efficient because it relies on users' help to decide whether the application leaks users' privacy. Zhao *et al.* [26] detect Android malwares based on the idea that most of the malware variants are created using automatic tools. Their approach statically extracts necessary features from each app and uses convolutional neural network to identify malwares, but it is not target for newly released malwares.

To reveal data leaks in apps and protect users' privacy, AppAudit [24] comprises a static API analysis that can effectively narrow down analysis scope and an innovative dynamic analysis which could efficiently execute application bytecode to prune false positive and confirm data leaks. AppIntent [25] detects the improper behavior that when a data transmission is not intended by the user, it is more likely a privacy leakage. It helps analysts to determine whether a data transmission is user-intended or not by providing a corresponding sequence of GUI manipulations. Apposcopy [9] presents a semantics-based approach for identifying a prevalent class of Android malware that steals private user information. MUDFLOW [2] learns "normal" flows of sensitive data from trusted applications to detect "abnormal" flows in possibly malicious applications. Leaksemantic [10] identifies suspicious sensitive network transmissions from mobile apps automatically. It utilizes machine learning classifiers to differentiate among the disclosures based on features derived from URLs in the traffic traces. These approaches focus on detecting data transmissions which are malicious or not intended by users, and they concentrate on revealing the process of data transmissions. Rubin *et al.* propose a technique [16] which focuses on detecting covert communications that no information is presented to the user neither on success nor on failure of the connection. In our work, we detect superfluous network transmissions by investigating the responses of network transmissions. Our approach studies the features of network communications and concludes the common points of how apps handle responses of superfluous network transmissions. Then we utilize the static information flow analysis to identify the superfluous network communications of which the responses are not used by the app. Overall, our work can be used as a complementary with existing researches.

7 Conclusion

The network transmission is an important way to exchange information between Android apps and remote servers for user required app functionalities. However, it is also used by improper behaviors to leak users' privacy. We propose a novel solution to detect superfluous network transmissions in Android applications. We take advantage of static information flow analysis to track how the responses of network transmissions are used by apps. The network transmissions are deemed superfluous if their responses are not utilized by apps. Our experimental results show that superfluous network transmissions are commonly existed in Android apps, and our approach can effectively detect superfluous network transmissions in Android apps.

Acknowledgments

The research is supported by National Natural Science Foundation of China under Grant No.61572453, No.61202404, No.61520106007, No.61170233, No.61232018, No.61572454, Natural Science in Colleges and Universities in Anhui Province under Grant No.KJ2015A257, and Anhui Provincial Natural Science Foundation under Grant No.1508085SQF215. The authors gratefully acknowledge the anonymous reviewers for their valuable comments.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM Digital Library*, vol. 49, no. 6, pp. 259–269, 2014.

[2] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering*, pp. 426–436, 2015.

[3] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 343–355, 2016.

[4] 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," in *NDSS*, TR-UCSB-2014-05, 2015.

[5] J. Cito, J. Rubin, P. Stanley-Marbell, and M. Rinard, "Battery-aware transformations in mobile applications," in *The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, pp. 702–707, 2016.

[6] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pp. 393-407, 2010.

[7] F-Droid, *F-Droid - Free and Open Source Android App Repository*, 2010-2018. (`https://f-droid.org/en/`)

[8] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[9] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 576–587, 2014.

[10] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, "Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications," in *IEEE Conference on Computer Communications, IEEE*, pp. 1–9, 2017.

[11] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *NDSS*, 2015. (`https://people.csail.mit.edu/rinard/paper/ndss15.droidsafe.pdf`)

[12] J. Huang, Y. Aafer, D. Perry, X. Zhang, and C. Tian, "Ui driven android application reduction," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 286–296, 2017.

[13] A. Kapratwar, *Static and Dynamic Analysis for Android Malware Detection*, 2016. (`https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=1488&context=etd_projects`)

[14] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A sealant for inter-app security holes in android," in *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*,, pp. 312–323, 2017.

[15] Q. Qian, J. Cai, M. Xie, and R. Zhang, "Malicious behavior analysis for android applications," *International Journal of Network Security*, vol. 18, no. 1, pp. 182–192, 2016.

[16] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard, "Covert communication in mobile applications (t)," in *The 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, pp. 647–657, 2015.

[17] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "Flexdroid: Enforcing in-app privilege separation in android," in *NDSS*, 2016. (`https://gts3.org/assets/papers/2016/seo:flexdroid.pdf`)

[18] M. Sun, T. Wei, and J. Lui, "TaintART: A practical multi-level information-flow tracking system for an-

droid runtime," in *Proceedings of ACM*, pp. 331-342, 2016. ISBN: 978-1-4503-4139-4.

[19] Tableau Software, *Number of Google Play Store Apps 2017 — Statistic*, 2017. (`https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`)

[20] E. Toch, Y. Wang, and L. F. Cranor, "Personalization and privacy: A survey of privacy risks and remedies in personalization-based systems," *User Modeling and User-Adapted Interaction*, vol. 22, no. 1-2, pp. 203–220, 2012.

[21] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 13, 1999.

[22] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 1329–1341, 2014.

[23] S. Wu, P. Wang, X. Li, and Y. Zhang, "Effective detection of android malware based on the usage of data flow apis and machine learning," *Information and Software Technology*, vol. 75, pp. 17–25, 2016.

[24] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *IEEE S&P*, 2015. ISBN: 978-1-4673-6949-7.

[25] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, pp. 1043–1054, 2013.

[26] Y. Zhao and Q. Qian, "Android malware identification through visual exploration of disassembly files," *International Journal of Network Security*, vol. 20, no. 6, pp. 1061–1073, 2018.

Biography

Jianmeng Huang received the B.S. degree in computer science from University of Science and Technology of China in 2013. He is currently working towards the Ph.D. degree at the Department of Computer Science and Technology, University of Science and Technology of China. His current research interests include information security and mobile computing.

Wenchao Huang received the B.S. and Ph.D degrees in computer science from University of Science and Technology of China in 2006 and 2011, respectively. He is an associate professor in School of Computer Science and Technology, University of Science and Technology of China. His current research interests include information security, trusted computing, formal methods and mobile computing.

Fuyou Miao received his Ph.D of computer science from University of Science and Technology of China in 2003. He is an associate professor in the School of Computer Science and Technology, University of Science and Technology of China. His research interests include applied cryptography, trusted computing and mobile computing.

Yan Xiong received the B.S., M.S., and Ph.D degrees from University of Science and Technology of China in 1983, 1986 and 1990 respectively. He is a professor in School of Computer Science and Technology, University of Science and Technology of China. His main research interests include distributed processing, mobile computing, computer network and information security.