Hindawi
Security and Communication Networks
Volume 2018, Article ID 2489214, 17 pages
https://doi.org/10.1155/2018/2489214

Research Article

Large-Scale Analysis of Remote Code Injection Attacks in Android Apps

Hyunwoo Choi and Yongdae Kim

KAIST, Daejeon, Republic of Korea

Correspondence should be addressed to Yongdae Kim; yongdaek@kaist.ac.kr

Received 14 July 2017; Revised 29 December 2017; Accepted 22 February 2018; Published 17 April 2018

Academic Editor: Po-Ching Lin

Copyright © 2018 Hyunwoo Choi and Yongdae Kim. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

It is pretty well known that insecure code updating procedures for Android allow remote code injection attack. However, other than codes, there are many resources in Android that have to be updated, such as temporary files, images, databases, and configurations (XML and JSON). Security of update procedures for these resources is largely unknown. This paper investigates general conditions for remote code injection attacks on these resources. Using this, we design and implement a static detection tool that automatically identifies apps that meet these conditions. We apply the detection tool to a large dataset comprising 9,054 apps, from three different types of datasets: official market, third-party market, and preinstalled apps. As a result, 97 apps were found to be potentially vulnerable, with 53 confirmed as vulnerable to remote code injection attacks.

1. Introduction

Widespread adoption of mobile technologies and significant increases in the number of smartphone users have resulted in a significant demand for diverse applications ("*apps*" for short). As of February 2016, over 65 billion apps have been downloaded from Google Play (the leading Android app store, with 80.7% market share [1]) [2], which had more than two million apps available for download up to March 2016 [3]. Further, to compete effectively in the highly competitive app market, developers are constantly adding new features to their apps.

With demands to support a plethora of features, apps often rely on external servers to dynamically update their resources at runtime. We generally refer to this class of updates as *dynamic resource update* (DRU). For example, apps often utilize *dynamic code loading* (DCL) to load additional code resources (e.g., .dex, .jar, .so, and .apk) at runtime to improve app startup performance, code reuse, extensibility, and self-updating [4]. In such cases, the additional codes could be downloaded from an external server while the app is running. In addition, an advertising library (AdSDK) included in an app may fetch ad resources such as images and videos from its servers and display them to the user so that advertisers can freely update their ads whenever they desire while the app is running. As recent statistics and studies show that over 44% of apps in Google Play include at least one mobile advertising library [5] and 32.48% of apps contain DCL components [6], DRUs are obviously prevalent in today's Android app implementations.

Unfortunately, several studies have revealed that DRUs are susceptible to remote code injection attacks [4, 6–8]. Apps that download external resources via an insecure protocol (such as HTTP) are vulnerable to man-in-the-middle (MITM) attacks. Consequently, it is possible for network attackers to modify or replace the DRU resources being downloaded. Falsina et al. [4] and Poeplau et al. [6] showed that if an app does not properly verify code resources downloaded via HTTP, an attacker can perform a remote code injection attack by injecting a malicious payload, which is then executed when the app loads the malicious payload at runtime. Watson [7] and Welton [8] found that remote code injection attacks can be carried out on other resource update procedures. If an app does not sanitize an input of ZIP extraction, filenames containing path traversal information may cause them to be stored or extracted outside of the intended directory. This situation can then be exploited by

attackers to overwrite existing arbitrary executables such as .so, .jar, and .dex files. Consequently, when attackers are able to modify ZIP archives being downloaded, they can perform remote code injection attacks.

From these studies [4, 6–8], we observe that there are three conditions that must be met for remote code injection attacks to be successful in Android apps: *no or bypassable validation checks*, *file overwrite vulnerabilities*, and *code trigger points*. The first condition includes the case when (1) apps do not perform integrity or authenticity checks on downloaded DRU resources or (2) attackers are able to bypass such validation checks. The second condition indicates the case when the injected payload can overwrite executables. The third condition is met when there exists a code trigger point where the overwritten files are loaded and executed in the app's context. Remote code injection attacks are successful when these three conditions are met.

Apps based on secure communication protocols (such as HTTPS) are not vulnerable to remote code injection attacks as an MITM attack is not possible unless vulnerabilities exist in an app's SSL/TLS implementation, such as trusting all certificates, allowing all hostnames, trusting many CAs, and mixed-mode/no SSL [9]. However, as shown in recent studies, the use of HTTP and improper use of HTTPS are widespread problems in Android apps [9–11], resulting in remote code injection attacks still being a serious threat in today's Android apps.

The problem becomes complicated when apps maintain multiple connections and download multiple DRU resources, because, in this case, all DRU updates have to be implemented securely. For example, apps generally apply HTTPS or integrity checking only to sensitive communications such as login, posting, purchasing, and self-updating activities and critical procedures. Remote code injection attacks can also be accomplished via other DRU resources such as images (.jpeg, .gif, etc.) and configurations (.xml, .json, .txt, etc.). Listing 1 is an example of vulnerable configurations. App developers may not be cognizant of the security implications of all DRU resources downloaded and stored in the file system. For example, developers usually implement the theme updates for apps by simply downloading images via HTTP. However, if file overwrite vulnerabilities and code trigger points exist in the theme updates, and there are no validation checks in update procedures, remote code injection attacks can still be carried out. While remote code injection attacks against code resource updates such as self-updates are well known, attacks against other resource update activities and their impacts are still largely unknown. In this paper, we show that updates involving other resources, such as archives, images, and temporary files, can also be vulnerable to remote code injection attacks. To this end, we first investigate general conditions for remote code injection attacks in Android apps and then present a static detection tool that automatically finds apps that satisfy these conditions. Our automatic detection tool uses the application binary (.apk) as input to identify potentially vulnerable apps using static program analysis. It combines network-aware program slicing, data dependency analysis, and string analysis to provide a comprehensive analysis of each condition. Finally we perform a large-scale analysis to identify vulnerable apps in the wild.

Our main contributions can be summarized as follows:

(i) We investigate three conditions for successful remote code injection attacks in Android apps: (1) no or bypassable validation checks, (2) file overwrite vulnerabilities, and (3) code trigger points.

(ii) We present the design and implementation of the *first* static detection tool that *automatically* identifies apps that meet these three conditions. More specifically, the detection tool takes only a binary (.apk), extracts DRU-related codes, and identifies whether the codes meet these three conditions by leveraging heuristics, string analysis, and data dependency analysis.

(iii) We perform a large-scale analysis using three different types of datasets, comprising 4,718 apps from an official market (Google Play), 2,967 from a third-party market (Tencent Myapp), and 1,369 from preinstalled apps (system apps). Our analysis identified a total of 97 apps as being potentially vulnerable, and 53 apps are confirmed to be vulnerable to remote code injection attacks.

The remainder of this paper is organized as follows: Section 2 provides necessary background, and Section 3 presents our threat model. Section 4 analyzes the three conditions necessary for successful remote code injection attacks. Section 5 presents our heuristics-based static detection tool. Section 6 presents the results of our large-scale analysis. Section 7 discusses mitigations and limitations. Section 8 reviews related work. Finally, Section 9 concludes this paper.

2. Background

This section provides background on dynamic resource updates and remote code injection attacks in Android apps.

2.1. Dynamic Resource Update. Android app developers often utilize external servers to dynamically update app resources while the app is running. As stated in the Introduction, throughout this work, we refer to this concept as *dynamic resource update* (DRU). DRUs are very commonly used for a variety of purposes.

2.1.1. Application Code Resource Update. At various times, Android apps may need to download additional features (i.e., application code) from external servers at runtime. For example, certain commercial apps such as game apps, which are initially distributed free of charge from an official market (such as Google Play) with minimum features, may need to provide premium features to their users after being purchased. In such cases, these game apps implement DRUs, in which the codes serving premium features are downloaded from the external server and then loaded into the app's context at runtime to provide the related premium service.

Another example of application code resource update is self-update. Self-update occurs when apps need to upgrade

```
GET http://w.mapbar.com/update/update/3he1/update.json
        ← 200 application/json
{
   "versionCode": 781172467,
   "size": 20006804,
   "url": "http://datamobile.mapbar.com/map/downloads/3in1/apk/MapbarNavi.apk",
   "mdvalue": "FF8AFDA9887E158F1FBF601489031AD1",
   "updateDescribe": [...]
}
```


Listing 1: An example of metadata in self-updating app.

themselves (e.g., .apk or .dex) or import libraries (e.g., .so). In self-update procedure, an app usually requests update information that specifies a downloadable URL from the external server. Listing 1 shows an example of update information in self-updating app. After receiving update information, the app downloads application code using the identified URL. Finally, the downloaded code will be loaded and executed for the purpose of self-updating. Unlike Google Play's update mechanism that requires user confirmation, self-update is usually executed without any user interaction. With the help of DCL (defined in the Introduction), apps can load downloaded code resources during execution. (Note that Google currently specifies the content policy of Google Play in its "Privacy and Security" section: *An app downloaded from Google Play may not modify, replace, or update itself using any method other than Google Play's update mechanism* [12].) However, Poeplau et al. [6] found that this policy is not technically enforced, to the extent that a large number of apps on Google Play still load external code resources.

2.1.2. Advertisement Resource Update. Recent statistics [5] show that advertising is prevalent in today's Android app implementations, where over 44% of apps in the Google Play include at least one mobile ad library. Once developers build their apps with AdSDK, it may fetch ad resources from its servers and display them to the users of the apps at runtime. In this manner, advertisers can freely update their ads whenever they wish to change the currently serving ads. (Note that ad resources are usually offered in the form of archives, such as ZIP or GZIP format, which may include a variety of compressed resources such as images, videos, HTML, and JavaScripts.)

Similarly, advertising libraries (such as .so files) can also be updated via external servers such as application code updates. When an app starts, the library checks its version using REST APIs (`HTTP` request, and `JSON` or `XML` response). If a new version is released, the library triggers its update logic to download the newly released library. After the library is successfully downloaded, it is loaded and executed in the context of the app.

2.1.3. Other Resources Updates. Apps also often need to download other types of resources from external servers to use at runtime. These resources can include temporary files, images, databases, and configurations needed in the context of the app. These are stored in the app's data folder (`/data/data/PACKAGE_NAME`) or an external storage (`/mnt` or `/sdcard`). For example, if an app needs to update a constant string, it may download an XML file from its server and save it to the data folder. The app then parses the constant strings from the XML file and adds them at runtime. (Note that whereas the Android framework defines certain types of application resources (Animation, Color State List, Drawable, Layout, Menu, String, Style, etc. [13]) that developers can provide in their resources directory (`res/`), the *"resource"* referred to in this paper includes not only those types defined in [13] but also a wide range of others (e.g., codes or archives) needed in app's context.)

2.2. Remote Code Injection Attacks. As stated in Section 1, apps communicating with external servers via a plaintext protocol such as `HTTP` are vulnerable to MITM attacks (note that apps using encrypted communications such as HTTPS can also be vulnerable to MITM attacks if they mis-implement TLS/SSL). Once an MITM attack is possible, an attacker can perform remote code injection attacks by modifying or replacing the resources being downloaded. From the literature [4, 6–8], we observe that remote code injection attacks can succeed under the following three conditions *(CI, CII,* and *CIII)*.

CI: No or Bypassable Validation Checks. There should be either no validation checks, or the validation checks can be bypassed by network attackers.

CII: File Overwrite Vulnerabilities. The injected payloads are stored in a specified location in accordance with the app's DRU implementations.

CIII: Code Trigger Points. The injected payload has to be executed in the context of the app when the app starts, or while it is running.

We investigate these three conditions in Section 4.

Example: A Remote Code Injection Attack against a Self-Updating App. Listing 1 shows an example of metadata in a self-updating app that is vulnerable to remote code injection attacks. In general, the self-updating app requests update

information from the server via HTTP and receives metadata containing update information such as *"url"* and *"mdvalue"* in JSON format. In this case, the self-updating app meets all the aforementioned conditions for a successful remote code injection attack. First, because the app communicates with the server via a plaintext protocol (HTTP), attackers can bypass its validation check by modifying the integrity value (*"mdvalue"* in the metadata) provided by the server *(CI)*. In addition, owing to the nature of the self-update mechanism, the downloaded .apk file is stored in the app's directory by replacing or overwriting the existing (legacy) executable *(CII)*, and the downloaded `.apk` is loaded and executed in the context of the app *(CIII)*.

In summary, with the aid of an MITM attack, if the app satisfies the three conditions above, attackers can easily perform a successful remote code injection attack by injecting their payload (`.apk`) with its correct hash value.

3. Threat Model

This section introduces the threat model used throughout this work.

We assume that apps are benign but potentially vulnerable to remote code injection attacks, that the external servers communicating with the apps are secure, so they cannot be compromised by an attacker, and also that users are benign and often connect to Wi-Fi networks consisting of unencrypted or untrusted access points (APs).

3.1. Ability of an Adversary. We assume that attackers cannot access a user's device but can obtain an app code running the user's device. The attackers cannot compromise or access the external server communicating with the app, but can install a rogue AP in a public or private area to lure the user into their Wi-Fi network [14]. Note that the rogue AP can be easily installed in such a manner that it has the same SSID as a trusted AP and has stronger signal strength than the original one. Attackers can also exploit a vulnerable AP to compromise it for mounting MITM attacks. For the remote code injection attack, attackers can inject their payload into transactions served over HTTP.

3.2. Attack Scenario. We consider the following attack scenario. We assume that the user initially connects to a Wi-Fi AP that is compromised or has been installed by an attacker, or the user connects to the attacker's device after connecting to the Wi-Fi APs, with the attacker performing ARP spoofing or any other techniques needed for an MITM attack such as DNS spoofing, so that the user's traffic passes through the attacker's device. When the user downloads DRU resources from an external server, the attacker carries out code injection by monitoring this transaction and replacing the resource being downloaded with his/her payload. When the injected payload is loaded and executed in the context of the app, the attacker gains access to the app's shell remotely and then can perform a privilege escalation attack to gain access to a higher level, such as root shell.

Note that our threat model is similar to prior works dealing with the security implications of *dynamic code loading* [4, 6]. However, the difference from prior works is that we only focus on network attackers and corresponding attack scenarios in which attackers can remotely inject payloads. Code injection attacks from malicious apps running on the same device are out of the scope of this paper.

4. Conditions for Successful Remote Code Injection Attacks

In this section, we analyze the three conditions required for successful remote code injection attacks against Android apps using decompiled code snippets as examples.

4.1. No or Bypassable Validation Checks. In general, apps usually ensure that downloaded resources have not been modified by using validation checks that verify the integrity of the downloaded resources. In this regard, servers provide a unique hash value produced by a hash function (such as MD5 and SHA-256) with the resource, and the app compares the provided hash value with a newly computed hash value for the downloaded resource. If the hash value that the app computes is the same as the hash value provided, the app determines that there is nothing wrong with the downloaded resource. Alternatively, servers can attach authenticity information (such as signatures) to the resource by digitally signing the produced hash value being distributed. This enables the app to check the downloaded resource by verifying its signature. Attackers could then tamper with the resource only if they are able to steal the server's signing key.

However, a hash value match does not necessarily guarantee that the downloaded resources have not been tampered by attackers. This is because if the provided hash value is transmitted via a plaintext protocol, with the aid of MITM attacks, attackers can bypass the validation checks by simply changing the hash value. Furthermore, and most importantly, developers often forget or do not recognize the importance of validation checks for downloaded resources [6]. For example, as discussed in Section 2, if a self-updating app does not verify the downloaded resource during the self-update procedure, attackers can successfully carry out a remote code injection attack by simply modifying the update information or by replacing the resources being downloaded.

Although the self-updating app verifies downloaded resources using a hash value provided by the corresponding server, this does not guarantee that the downloaded resources have not been modified by attackers. For example, in Listing 1, the self-updating app verifies the downloaded resource by examining whether the provided MD5 hash value *("mdvalue": "FF8AFDA9887E158F1FBF601489031AD1")* is equal to the hash value computed by the app. In this case, if the hash value is transmitted via a plaintext protocol (HTTP), with the aid of MITM attacks, attackers can bypass the validation check by modifying the hash value (*"mdvalue"* in Listing 1). Therefore, remote code injection attacks can still be successful in cases where the validation checks are bypassed.

```
(1)  protected java.lang.String doInBackground(java.lang.Void[]) {
(2)    . . .
(3)    URL url = new URL("http://www.appnext.com/android/images2.zip");
(4)    HttpURLConnection conn = (HttpURLConnection)url.openConnection()
(5)    int length = conn.getContentLength();
(6)    byte[] buf = new byte[length];
(7)    ...
(8)    DataInputStream dis = new DataInputStream(url.openStream());
(9)    dis.readFully(buf);
(10)   . . .
(11)   File dir = getDir("appnext", MODE_PRIVATE);
(12)   File file = new File(dir, "images2.zip")
(13)   . . .
(14)   DataOutputStream dataOut =
                new DataOutputStream(new FileOutputStream(file));
(15)   dataOut.write(buf);
(16)   . . .
(17)   String path = getFilesDir().getAbsolutePath().append("/appnext/");
(18)   unzip(file.getAbsolutePath(), path)
(19)   ...
(20) }
```

Listing 2: Resource download without validation checks (decompiled from Appnext SDK [15]).

Another example in which the DRU resource is downloaded without validation check is illustrated in Listing 2. The code snippets give the details of an image being downloaded in Appnext SDK [15], a mobile monetization and app distribution platform. The SDK downloads image files in the form of ZIP archives from an external server with a fixed download link *"http://www.appnext.com/android/images2.zip"* (line (3) in Listing 2), which is hardcoded within the app. Then, it stores the downloaded ZIP archive in the *"appnext"* directory without any validation checks. In this case, if the DRU can be abused to overwrite the existing executables, attackers can successfully carry out a remote code inject attack. Note that DRUs such as image resource update occur with very high frequency in today's app implementations.

4.2. File Overwrite Vulnerabilities. After attackers inject their payloads bypassing the validation checks for resources, the injected payloads are stored in a specified location in accordance with the app's DRU implementations, usually in the app's data directory (`/data/data/PACKAGE_NAME`) or in external storage (such as an SD card). If the DRU that an attacker targets is the application code update, the injected code is replaced with the existing code resource (e.g., .dex, .jar, or .so) and then loaded and executed when the app triggers the update logic. In such cases, the attacker only needs to inject the payload without any considerations to carry out a successful remote code injection attack.

On the other hand, there could be no DCLs in the app, which is common in the majority of apps. However, attackers can still successfully perform remote code injection attacks by means of file overwrite vulnerabilities. If there is an arbitrary write vulnerability in the app, attackers can exploit it to overwrite the existing executables such as .dex, .so, and .jar. Watson [7] and Welton [8] showed that an unsafe ZIP extraction can be used for arbitrary write vulnerability; thus, remote code injection attacks can be successful with the aid of file overwrite vulnerabilities.

In this section, we further analyze other file overwrite vulnerabilities that can be used for remote code injection attacks.

Unsafe ZIP Extraction. Android apps often implement ZIP archives to efficiently download or upload resource files from/to their external servers over the network. However, as shown by Watson [7] and Welton [8], if developers do not consider the security implications of unsafe ZIP extractions [16], arbitrary overwriting vulnerabilities that allow attackers to overwrite the existing files with their injected payloads may be present.

Listing 3 shows an example of an unsafe ZIP extraction implemented in Appnext SDK [15]. The Appnext SDK updates its image resources by directly downloading a ZIP archive (`images2.zip`) that contains multiple .png files from its server (`doInBackground()` in Listing 2). After downloading the ZIP archive, the Appnext SDK decompresses the downloaded ZIP archive, which is stored in the app's internal directory (`/data/data/PACKAGE_NAME/appnext/image2.zip`), so that the image files can be located in the intended directory (`/data/data/PACKAGE_NAME/appnext/files/`). To achieve this, the `unzip()` method first gets a filename (line (6)), which it then uses to create a file output stream to write to the image file (line (14)). Therefore, if the input of filename is not sanitized and contains path traversal information, it may cause the file to be extracted outside of the intended directory. For example, if the filename is

```
(1)  public void unzip(String str1, String str2) {
(2)      . . .
(3)      Object obj = new ZipInputStream(new FileInputStream(str1));
(4)      String str3 = ((ZipInputStream)obj).getNextEntry();
(5)      . . .
(6)      String str4 = str2 + str3.getName();
(7)      if (!paramString1.isDirectory()) {
(8)          extractFile((ZipInputStream)obj, str4);
(9)      }
(10)         . . .
(11) }
(12)
(13) private void extractFile(ZipInputStream is, String str) {
(14)     BufferedOutputStream out =
                 new BufferedOutputStream(new FileOutputStream(str));
(15)     byte[] arrayOfByte = new byte[4096];
(16)     . . .
(17)     int i = is.read(arrayOfByte);
(18)     out.write(arrayOfByte, 0, i);
(19)     . . .
(20) }
```

Listing 3: Unsafe ZIP extraction (decompiled from Appnext SDK).

"./../../../data/data/PACKAGE_NAME/files/target.so," the `target.so` file would be replaced. Attackers can exploit this fact to overwrite the existing arbitrary code resources with their injected payload. Note that this vulnerability occurs when the app does not sanitize the input of the pathname or verity its location.

Unsafe Content-Disposition Implementation. Modern web browsers often utilize an HTTP header to forcefully download an external resource instead of rendering it on the browser. To forcefully download with the HTTP header, the server adds a Content-Disposition field that includes a filename parameter in the HTTP response header (line (3) in Listing 4) and, during the downloading of the external resource on the client side, the browser retrieves a filename from the HTTP response header and stores the downloaded resource with the provided filename.

Android developers often use the HTTP response header as metadata to retrieve the information of the downloaded resource. For example, an app can retrieve the filename of the downloaded resource from the HTTP response header to display the resource name on a screen or save the downloaded resource into internal storage. To do this, the app first obtains the value of the Content-Disposition field using network APIs such as `org.apache.http.HttpResponse.getFirstHeader()` and the `java.net.HttpURLConnection.getHeaderField()` method. The app then parses the filename using a dedicated API such as `guessFileName()` in the `android.webkit.URLUtil` class or a user-defined parser that implements a regular expression match. In this case, however, if the app does not properly parse the filename and simply uses this as a filename to create a file, an arbitrary overwriting vulnerability may exist. For example, when using regular expression matching with the pattern string *attachment; "s*filename "s*="s*""([^""]*)""*, if the matcher evaluates the *attachment; filename="./../../../target"* string, the matcher would find a match in *"./../../../target"* string that contains path traversal information. Thus, as with the unsafe ZIP extraction, attackers can overwrite the arbitrary files by modifying the Content-Disposition field in the HTTP header.

4.3. Code Trigger Points. To successfully carry out a remote code injection attack, the injected payload has to be executed in the context of the app when the app starts, or while it is running. Therefore, the attacker has to identify a code trigger point that loads the injected payload and executes it. A self-update is a good example of code containing a code trigger point, by which the payload is loaded and executed after downloading newly released code.

In this subsection, we investigate possible code trigger points that can be used for remote code injection attacks.

Runtime Library. Android apps can include runtimes libraries (such as .jar or .so file), which are loaded when the app starts or while the app is running by using `loadLibrary` (in case of .so) or `DexClassLoader` (in case of .jar) method. The Native Development Kit (NDK) allows developers to build their own C/C++ source code, or to take advantage of prebuilt libraries. Developers can utilize the native libraries to improve the app's performance or reuse their own or another developer's libraries. In addition, developers can load classes from .jar or .apk to execute methods that are not contained as part of an application. These runtime libraries can be used as a target for code trigger points. In Android,

```
HTTP Response Header
(1) 200 OK
(2) Content-Type: text/html; charset=utf-8
(3) Content-Disposition: attachment; filename="filename.txt"
(4) Content-Length: 22
```

Listing 4: Content-Disposition field in HTTP header.

when an app uses native libraries, which is built by Android NDK, these libraries are stored in `/lib` directory and have system privilege. Therefore, native libraries in `/lib` cannot be used for trigger points. However, when developers create libraries (including .so and .jar) and mark them as writeable and put them in the app's `/assets` directory, these libraries can be considered as potential trigger points. After the app is installed, these libraries can be stored in the app's internal directory such as `/data/data/PACKAGE_NAME/files`. Attackers who know this path information can overwrite one of these libraries to execute their injected payload.

Multidex. The Android platform supports a multidex to deal with the 64k reference limit that limits the total number of methods that can be invoked within a single DEX file to 65,536, including Android framework methods, library methods, and user-defined methods [17]. To support the multidex, during build time, an Android build tool constructs a primary dex (`classes.dex`) and other secondary dexes (e.g., `classes2.dex` and `classes3.dex`) as needed and packages them into a .apk file for distribution. While installing the app, the secondary dexes are extracted into the `/data/data/PACKAGE_NAME/code_cache/secondary-dexes/` directory and loaded when the app starts. For example, the Runtastic [18] app containing the multidex (`classes2.dex`) extracts the secondary dex into the corresponding directory and rename it as `com.runtastic.android-1.apk.classes2.dex`. Therefore, if attackers can overwrite this secondary dex, they can trigger their injected payload when the app starts.

Runtime.exec(). As with Java applications (an application cannot create an instance of the `Runtime` class, but can obtain an instance by invoking the `getRuntime()` method), Android apps can also get an instance of the `Runtime` class by invoking the `getRuntime()` method. Using the `exec()` method of the `Runtime` class, the apps can execute arbitrary executables in a separate native process by simply providing the specified shell command as an argument. This operates similar to Linux's `system()` method, and thus Android app developers often utilize this method for easy implementation. However, the `exec()` method of the `Runtime` class can be used for the code trigger point to launch remote code injection attacks. If an attacker knows the argument to be passed to the `exec()` method and replaces the existing file that the argument points to, she can execute her payload.

For example, Umeng PushSDK [19], one of China's popular push notification services, implements the `exec()` method to provide their push notification service through a shell command. Specifically, it creates an instance of `Process` by invoking `Runtime.getRuntime().exec("sh")` and redirects its data stream to `DataInputStream/DataOutputStream` instances so that the app can execute the commands by writing to `DataOutputStream` or reading from `DataInputStream`. When the app starts, the push library checks whether a *ServerDaemon* file exists in the app's files directory (`/data/data/APP_PACKAGE_NAME/files/`), and if it exists, it executes the file through the `exec()` method with the specified arguments. In the case where the *ServerDaemon* file does not exist in the files folder, the library creates a new *DaemonServer* file and then executes it as well. Attackers can take advantage of this fact to identify the code trigger point; that is, they can execute their injected codes by overwriting the target file passed as a parameter of the `exec()` method, such as an example of the *DaemonServer* file.

5. Automatic Detection

In order to detect apps that are potentially vulnerable to remote code injection attacks, we developed a static analysis tool (https://gitlab.com/zemis0ls0l/remote code injection attack) that automatically identifies code snippets that meet the three conditions described in Section 4. In this section, we outline the design and implementation of our static detection tool.

5.1. Overview. Figure 1 illustrates the three main components of our detection tool: *preprocessor*, *program slicer*, and *vulnerability checker*. At a high level, our static detection tool takes a .apk file as input and converts it to Jimple (Jimple is a popular intermediate language based on three components per statement in code that is often used for bytecode optimization) intermediate representation by means of Soot [20], a static analysis framework that provides Jimple for both Java and Android (Soot framework includes Dexpler [21] that converts Dalvik bytecode to Jimple) and call graph analysis. Then, based on program slicing [22] with interesting points (i.e., APIs) and heuristics, the detection tool analyzes DRU-related code to identify code snippets that meet the three conditions. The output of the tool includes a set of information that can be used to identify whether the app is vulnerable to remote code injection attacks. Note that the detection tool operates on top of Jimple and does not require the source code of the app to be analyzed.

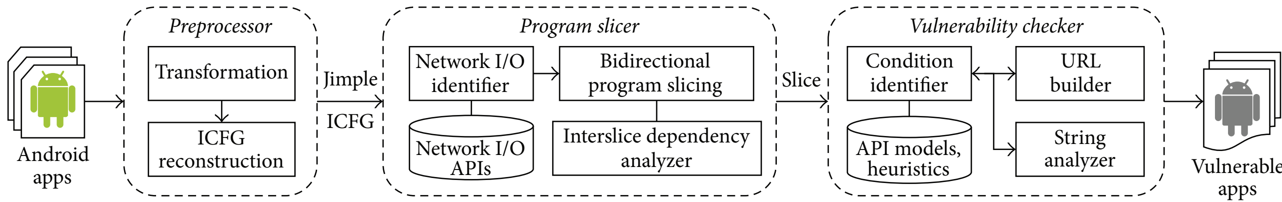


Figure 1: Design overview of the static detection tool.

5.2. Preprocessor. As with other state-of-the-art static analysis studies for Android apps (such as Bartel et al. [23], Geneiatakis et al. [24], Woodpecker [25], Chex [26], and Paddyfrog [27]), our detection tool first translates Dalvik bytecode to an intermediate representation and then constructs an interprocedural control-flow graph (ICFG) (it is also known as a super control-flow graph (sCFG)) representing all possible execution paths of an app, for a given .apk file. Because the accuracy of static analysis relies on the control-flow graph, a precise ICFG needs to be constructed in order to improve the precision of static analysis. However, unlike Java applications, because Android apps are framework-based as well as event-driven, generating the corresponding ICFG is challenging. For example, instead of a main method, Android apps contain many entry points that are implicitly called by the Android framework. In addition, the Android framework allows apps to register various types of callbacks, which are also invoked by the framework. This means that code snippets contained in callback methods cannot be analyzed without recognizing such implicit edges because an incorrect ICFG does not have an outgoing control-flow edge to the callback method.

In this work, to construct precise ICFGs, we leverage FlowDroid [28], a static taint analysis framework that provides flow- and context-sensitive and interprocedural data flow analysis for Android apps. FlowDroid models the component lifecycle of the Android framework and incrementally reconstructs the control-flow graph when identifying newly discovered callback methods. Unfortunately, FlowDroid does not support identifying thread-related classes in Android apps (including `AsyncTask`, `Thread`, and `Runnable`) that generate implicit control flows through callbacks. However, in Android, resource download tasks are generally implemented by utilizing threading classes because network operations cannot be run on the main thread (developers can use `Thread`, `AsyncTask` for short-running tasks and `Service` for long-running tasks to do network operations). Thus, identification of thread-related edges is necessary for our detection tool. To correctly analyze the apps, we add extensions to support such threading classes. For example, to support the `AsyncTask` class, we identify all `AsyncTask` instances and augment the call graph by adding edges that connect to the `AsyncTask` instance. This can be accomplished by replacing the invocation of `execute()` with invoke calls to `onPreExecute()`, `doInBackground()`, and `onPostExecute()`. Finally, we reconstruct the ICFG by combining FlowDroid's incremental CFG construction with our extensions. The reconstructed ICFG is used for the next bidirectional program slicing step.

5.3. Program Slicer. In order to extract the Jimple slices, we implement a forward and backward slicing algorithm (see Appendix A), which are based on the network-aware program slicing approach proposed by Choi et al. [29, 30]. The slicing algorithm works bidirectionally based on the ICFG constructed in the previous step. The algorithm starts at the predefined interesting points (network I/O APIs and its parameters) such as `java.net.URL.openConnection()` and `org.apache.http.client.HttpClient.execute()` by adding the variables of these points to a worklist. Then, it walks forward and backward on the ICFG while analyzing the data dependency between the variables in the worklist and the current variable in the Jimple statements. For interprocedural forward/backward analysis, our algorithm keeps a call stack that records the current method (i.e., caller) and its location. In this way, the program slicer provides a context-sensitive data flow analysis. The program slicer continues this analysis recursively in this manner. The analysis terminates when there is no entry in the worklist, when it reaches the entry point of the app (in the case of backward slicing), or when it encounters sink APIs (in the case of forward slicing) such as `FileOutputStream.write()`.

For example, in Listing 2, the program slicer starts at `java.net.URL.openConnection()` and walks forward and backward while analyzing the data dependency. The backward slicing terminates at line (3) in Listing 2, and the forward slicing terminates at line (18) in Listing 3. Note that, for readability, we used Java code instead of Jimple IR and listed only the results of the program slicing in the code snippets.

Once the program slicer has extracted slices, the interslice dependency analyzer identifies dependencies between the extracted slices. The goal of this analysis is to identify any dependencies between HTTP response and HTTP request. Figure 2 shows an example of how the interslice dependency analysis operates on request (`requestA` and `requestB`) and response (`responseA` and `responseB`) slices. In the figure, an app receives metadata (line (5) in `requestA`) and then parses it (line (2) in `responseA`) to obtain a resource download URL. Then, using the obtained URL, the app downloads and stores the resource (line (2) in `requestB` and line (3) in `responseB`, resp.). In this case, a dependency exists between `responseA` and `requestB`. To identify this, we leverage the taint-based approach proposed by Choi et al. [29, 30], in which the dependency is determined by identifying the data flow from the source (line (1) in `responseA`) to the sink (line (1) in `requestB`).

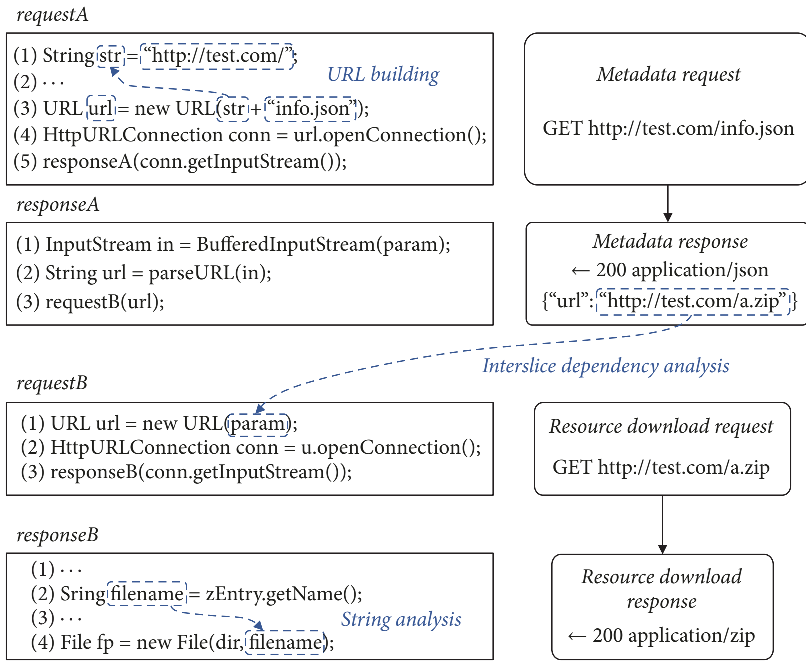


Figure 2: URL building, string analysis, and interslice dependency analysis.

Finally, the program slicer records Jimple slices consisting of HTTP request/response and its dependencies for the next step, in which code snippets meeting the three conditions for vulnerability to remote code injection attacks are identified.

5.4. Vulnerability Checker. After the program slicer extracts all the slices that affect network operations, as well as their dependencies (HTTP request and response), the vulnerability checker identifies code snippets that meet the three conditions for remote code injection attack vulnerability. The vulnerability checker achieves this by implementing the following heuristics. Note that the vulnerability checker utilizes FlowDroid's taint analysis that provides flow- and context-sensitive and interprocedural data flow analysis.

(1) Identifying No or Bypassable Validation Checks. To identify instances where there are no validation checks, we find where the message digest class is used, such as `java.security.MessageDigest`, from the given response slices. If there is no use of invocation of message digest methods such as `update()` or `digest()`, we consider that no validation checks exist. However, although the app validates the resource using a provided hash value, if the hash value is transmitted via HTTP, the validation check can be bypassed as described in Section 4. Consequently, to identify bypassable validation checks, we utilize a URL builder that generates a URL that feeds into the network I/O APIs given HTTP request slices produced via backward slicing. The URL builder models high-level Java and Android APIs such as `append()` and `toString()`, which are often used for string manipulations (we further describe the URL builder below). If the generated URL is HTTP, we consider the validation checks to be bypassed even if the message digest functions are present in the slices.

(2) Detecting File Overwrite Vulnerabilities. Given HTTP response slices produced by forward slicing, to identify an unsafe ZIP extraction, we first find an instance of `java.util.zip.ZipInputStream` and file classes such as `java.io.File`. We then check whether the code validates the name of each entry before extracting it. To achieve this, the string analyzer randomly assigns an initial string that contains path traversal information (e.g., "*./../../../target*") as the value of `java.util.zip.ZipEntry.getName()`, and then the string analyzer tracks slices while updating its manipulations (based on API models) until it encounters the file method. When the initial string is passed to the parameter in the file method, if the path traversal information of the initial string does not change (or is filtered out), we consider it to be file overwrite vulnerabilities; that is, it is an unsafe ZIP extraction.

Similarly, in order to identify an unsafe Content-Disposition implementation, we first find a method invocation (either `org.apache.http.HttpResponse.getFirstHeader()` or `java.net.HttpURLConnection.getHeaderField()`) commonly used for parsing values in HTTP headers, and file write methods, and then check whether the code properly parses the filename from the Content-Disposition field. Either of two methods can be used to parse a filename from the Content-Disposition field: string manipulation APIs or regular expression matching. The first method usually finds a certain string using `indexOf()` and splits the string using `subString()`. Thus, the same approach used to find unsafe ZIP extractions can be leveraged to model these string manipulation APIs

using the string analyzer. The second method parses the filename by matching a regular expression. To facilitate this method, we extract a constant string that is passed to the parameter in `java.util.regex.Pattern.compile()` and then evaluate this pattern string with our test string (*"./../../../../../target.so"*). After matching the regular expression, if the result is the same as our initial test string, we consider it to be a file overwrite vulnerability as well.

(3) Identifying Trigger Points. To identify trigger points, we utilize three different properties of Android apps. Identifying multidex is straightforward. Android apps contain dex files (.dex) in root directory inside a .apk file, which is a ZIP archive format. Thus, we can easily identify multidex by decompressing the `.apk` file and then checking whether secondary dex files (such as `classes2.dex`) exist. Unlike the multidex, identifying runtime library and Runtime.exec() need string analysis. The string analyzer starts by detecting `dalvik.system.DexClassLoader()`, `java.lang.System.loadLibrary()`, and `java.lang.Runtime.exec()` methods. If method invocations are found, the string analyzer performs backward analysis with respect to the parameter value of the method in order to build a constant string. Once the constant string is generated, the string analyzer examines it by comparing it with known executables. In case of runtime library, we compare the constant string with the names of executables, which are extracted in `/assets` directory inside the .apk file. If there is a match, we consider it contains a trigger point. In case of Runtime.exec(), we filter out cases in which the generated constant string contains one of the executables located in the "`/bin`" and "`/sbin`" directories. After filtering, if there are any remaining constant strings, we consider it as a trigger point. In this way, we can identify the possible trigger points.

(4) Distinguishing Whether the Communication Protocol Is Secure. Even though apps may meet the three conditions identified for successful remote code injection attacks, if the apps use the HTTPS protocol, they may not be vulnerable to remote code injection attacks. Thus, we further analyze the apps to identify whether they use HTTPS. To this end, we implemented a URL builder module that generates URLs by walking back from a URL initializer such as `java.net.URL.URL()` or the `org.apache.http.client.methods.HttpGet()` method. During the backward analysis, the URL builder models string manipulation APIs such as `java.lang.StringBuilder.append` where the constant strings are appended to build a full URL. In addition, the URL builder handles references to resource objects, such as `Android.R`, whose values are stored as user-defined files in the .apk (e.g., `res/values/strings.xml`). On generation of the full URL, the URL builder distinguishes between a *static URL*, which is hardcoded in the codes, and a *dynamic URL*, which comes from other network inputs. This is achieved by identifying the results of interslice dependency analysis (using the interslice dependency analyzer in Section 4). In the case of the static URL, if the URL starts with *"HTTP://,"* we consider it vulnerable to remote code injection attacks. However, in the case of dynamic URL, because we cannot identify dynamically generated URLs, owing to the limitation of static analysis, we consider it potentially vulnerable.

On the other hand, there are cases where the HTTPS is not properly implemented, meaning that attackers exploit mis-implemented HTTPS to carry out MITM attacks. To deal with such cases, using Mallodroid by Fahl et al. [9], we identify the mis-implemented HTTPS, such as trusting all certificates or allowing all hostnames. If mis-implemented HTTPS is found, we also consider it vulnerable to remote code injection attacks.

6. Large-Scale Analysis

In order to assess the current state of vulnerable apps to remote code injection attacks in the wild, we applied our detection tool to three different types of datasets. In this section, we first describe the three datasets analyzed and then present the results of our large-scale analysis for each dataset.

6.1. The Datasets. To evaluate our detection tool and identify apps potentially vulnerable to remote code injection attacks, we collected three different datasets comprising thousands of apps. Table 1 gives a summary of the datasets. The first dataset is an official market dataset (Google Play), the second is a third-party market dataset (Tencent Myapp [31]), and the third is a system application dataset extracted from manufacturers' firmware images, including Samsung [32] and Huawei [33].

More details can be found in Appendix B.

6.2. Results. As described in Section 5, even where an app has file overwrite vulnerabilities, it is not necessarily vulnerable to remote code injection attacks, because if the app uses HTTPS properly, attackers cannot perform MITM attacks to inject their payload. Therefore, whether the app is vulnerable to remote code injection attacks depends on the request protocol (i.e., the URL string); URL strings starting with *"http://(.∗)"* are vulnerable whereas those starting with *"https://(.∗)"* are not. In addition, if the URL is from another HTTP transaction (in the case of dynamic URLs), we also cannot identify whether the app is vulnerable, because the URL string could not be determined via static analysis. For this reason, we divided the results into two groups: ❶ potentially vulnerable apps and ❷ flagged vulnerable apps. Potentially vulnerable apps contain a static HTTP URL and a dynamic URL, whereas flagged vulnerable apps contain only a static HTTP URL. Note that a mis-implemented HTTPS can also be vulnerable to MITM; flagged vulnerable apps contain HTTPS mis-implementation as well.

Official Market (Google Play). Tables 2 and 3 show the results obtained for the Google Play market dataset. Using our detection tool, we analyzed 4,718 diverse apps from 28 categories. Table 2 shows the number of vulnerable apps that met two conditions, CI (no validation check or bypassable validation check) and CII (arbitrary overwriting

Table 1: Summary of dataset.

Dataset	Type	Number of applications
Google Play	Official market	4,718
Tencent Myapp	Third-party market	2,967
System apps	Preinstalled apps	1,369
Total number of applications		9,054

Table 2: Results of Google Play dataset ($CI \cap CII$). All vulnerable apps were manually confirmed.

Category	Type	❶ Number of potentially vuln apps	❷ Number of flagged vuln apps
File overwrite vulnerabilities	Unsafe ZIP	75 (1.5%)	49
	Unsafe Content-Disposition	15 (0.3%)	0

❶ = static HTTP URL + dynamic URL; ❷ = ❶− dynamic URL.

Table 3: Results of Google Play dataset *(CIII)*.

Category	Type	❸ Number of apps in this class	❹ Number of flagged vuln apps (❷ ∩ ❸)
Code trigger points	Runtime library	188 (3.9%)	0
	Multidex	631 (13.3%)	17
	Runtime.exec()	174 (3.6%)	22

vulnerability). We found that 90 apps (1.9%) contained file overwrite vulnerabilities, of which 75 (1.5%) implemented the decompressing ZIP archives in an unsafe manner and 15 (0.3%) contained unsafe Content-Disposition implementations. After excluding the dynamic URLs, we found 49 flagged vulnerable apps, in which all apps implementing the unsafe Content-Disposition used only dynamic URLs.

Table 3 shows the number of apps that satisfied CIII (trigger point). In the table, 188 (3.9%) of the 4,718 apps contain the runtime libraries, 631 (13.3%) contain secondary dex files, and 173 contain the Runtime.exec(). Among them, 39 (= 0 + 17 + 22) apps contain file overwrite vulnerabilities (i.e., meeting all conditions, CI ∩ CII ∩ CIII). Finally, after removing multiple trigger points, we consequently obtained 25 apps vulnerable to remote code injection attack in the Google Play dataset. Particularly, some of vulnerable apps that we found are extremely popular such as Opera browser, Pandora Radio (with more than 500,000,000 downloads), and CM Locker Repair Privacy Risks (with more than 100,000,000 downloads).

Third-Party Market (Tencent Myapp). Tables 4 and 5 show the results for the Tencent Myapp market dataset. We analyzed 2,967 apps (from 29 categories). As shown in Table 4, we found 82 apps (2.7%) that satisfied CI and CII, that is, containing no or bypassable validation checks and file overwrite vulnerabilities. More specifically, 72 apps (2.4%) contained unsafe ZIP extraction, and 10 (0.3%) contained unsafe Content-Disposition implementation. This rate is almost twice that of the Google Play marketplace. After ruling out dynamic URLs, we identified 45 flagged vulnerable apps, 43 of which contained unsafe ZIP extraction and the remaining two containing unsafe Content-Disposition implementation. In addition, Table 5 shows the number of trigger points in the dataset. In the third-party dataset, 1,828 apps (61.6%) contain runtime libraries, which is much more than the Google Play dataset. Other rates of trigger point were 440 (14.8%) for multidex and 368 (12.4%) for the Runtime.exec(). The number of apps with trigger point, no or bypassable validation checks, and file overwrite vulnerability simultaneously present was 20 for runtime library, 6 for Multidex, and 12 for Runtime.exec(). After removing multiple trigger points, we found 28 vulnerable apps, including extremely popular apps such as `com.tencent.qqlive` (1,200,000,000 downloads), `com.baidu.BaiduMap` (770,000,000 downloads), `cn.kuwo.player` (470,000,000 downloads), `cn.eclicks.wzsearch` (111,000,000 downloads), and `com.og.danjiddz` (32,370,000 downloads).

Preinstalled Apps. In Android, preinstalled apps are generally granted many more capabilities with critical permissions than a normal app. This means that if a preinstalled app is vulnerable to remote code injection attacks, attackers can have a much greater impact on the device. For example, an app holding an `INSTALL_PACKAGES` permission can install another .apk silently. Therefore, if an attacker successfully gains target app's privilege by performing remote code injection attack, s/he could use the `INSTALL_PACKAGES` permission to install malware on the device without any difficulty. Table 6 shows the vulnerable apps results from the system app dataset. In our study, two preinstalled apps that met two conditions (CI ∩ CII) for successful remote code injection attacks were found. The WildTangent Game app (http://www.wildtangent.com/), which is included in AT&T's Galaxy S6 (SM-G890A) model, contains an unsafe ZIP extraction and uses a dynamic

Table 4: Results of Tencent Myapp dataset ($CI \cap CII$).

Category	Type	Number of potentially vuln apps	Number of flagged vuln apps
File overwrite vulnerability	Unsafe ZIP extraction	72 (2.4%)	43
	Unsafe Content-Disposition implementation	10 (0.3%)	2

Table 5: Results of Tencent Myapp dataset *(CIII)*.

Category	Type	Number of apps in this class	Number of flagged vuln apps
Code trigger points	Runtime library	1,828 (61.6%)	20
	Multidex	440 (14.8%)	6
	Runtime.exec()	368 (12.4%)	12

Table 6: Results for system apps dataset (preinstalled apps).

App (.apk)	Type	HTTP(S)	Model	Carrier
ZinioReader	Unsafe ZIP extraction	HTTP (static)	SGH-T769 (Galaxy S)	T-Mobile
WildTangent_ATT	Unsafe ZIP extraction	HTTP (dynamic)	SM-G890A (Galaxy S6)	AT&T

HTTP URL. Consequently, if there is a trigger point (we did not find possible trigger points), attackers can take control of the device by injecting and overwriting their payload. Meanwhile, because the WildTangent Game app has `INSTALL_PACKAGES` permission, after taking control of the device, attackers can install malware silently. Zinio Reader (https://gb.zinio.com/www/apps/desktop.jsp#/download), a magazine app included in the Galaxy S device (SGH-T769) of T-Mobile, also contains unsafe ZIP extraction. However, unlike the WildTangent Game app, the Zinio Reader app contains a static HTTP URL and does not have `INSTALL_PACKAGES` permission.

Vulnerable Libraries (SDKs). A vulnerable Android SDK is very serious as a single vulnerability can be present in a large number of apps and developers usually do not focus on the security implications of these SDKs. In our results, we identified four popular vulnerable SDKs that have unsafe ZIP extraction. Specifically, these vulnerable SDKs include three Ad libraries (AppNext (http://www.appnext.com/), AdMarvel (http://www.admarvel.com/), and Madhouse SmartMAD (http://www.smartmad.com/)) and one social library (QQ SDK). To measure how many apps can be affected by these vulnerable SDKs, we identified the proportion of vulnerable apps by referring to recent statistics [5]. As these libraries have around 0.91% market share overall (approximately 300,000 apps in Google Play), we can approximately guess the proportion of vulnerable apps in the wild.

Dangerous Permissions. Once an attacker carries out a remote code injection attack by exploiting the three conditions, the attacker obtains the target app's privilege to more effectively perform sensitive operations (e.g., tasks that can cost money or access private user data). In our threat model, we assumed that after gaining the app's privilege the attacker can additionally perform a privilege escalation attack to obtain higher privilege (i.e., system or root). However, if the target device does not have such vulnerabilities that can be used for privilege escalation attacks, the attacker can only do limited operations (i.e., attacks) such as reading contacts, sending SMS, or getting location information, according to the permissions specified in the manifest of the app.

In our evaluation, we further analyzed flagged vulnerable apps to identify what operations can be conducted by attackers even without escalating local privilege. To do this, in accordance with the Android Developers Guide [34], we first categorized permissions into nine permission groups (because the vulnerable apps that we found did not have sensor permissions, we ruled out the `SENSORS` permission group in the graph) and then counted the number of permissions belonging to each group. Figure 3 shows the number of dangerous permissions associated with each group. From the results, we found that most of vulnerable apps have at least `PHONE`, `LOCATION`, or `STORAGE` permission. This means that once an attacker gains the app's privilege on performing a remote code injection attack, s/he can attempt various attacks even without system level privilege. For example, with `CALL_PHONE` permission, the attacker may make premium phone calls in the background, an overbilling attack [35]. The attacker can also perform a race condition attack via a shared SD card directory to install malware when the app has `WRITE_EXTERNAL_STORAGE` permission [6].

Manual Review. Finally, we manually review 97 apps which are confirmed vulnerable or potentially vulnerable (53 apps) in the static analysis tool. Given these results, we first decompile the apps and identify the DRU code points that the tool reported. Then, we verify whether the code snippets indeed met the aforementioned three conditions. We also test whether the code snippets are reachable from the entry points. The more complex the app is, the more time-consuming and challenging such a review process becomes. For that reason, we also install the app and run it on an emulator to complement the static analysis. In this way, we

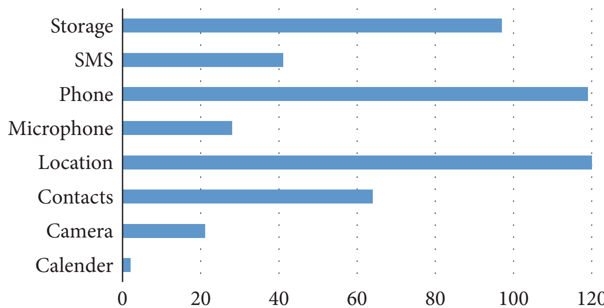


Figure 3: Number of dangerous permissions.

take a best-effort approach in combination with static and dynamic analysis. In the end, we confirmed that the reported apps are all vulnerable to remote code injection attacks.

7. Mitigations and Limitations

In this section, we discuss how remote code injection attacks in Android apps can be mitigated and the limitations of our static detection tool.

7.1. Mitigations

7.1. Mitigations. Mitigation can be considered from two perspectives: the app developer level (filename sanitization and secure communication protocol) and the framework level (secure code execution).

Filename Sanitization. As described above, filenames containing path traversal information may cause them to be stored or extracted outside of the intended directory. Attackers can exploit this vulnerability by overwriting the existing arbitrary executables. To defend against remote code injection attacks caused by file overwrite vulnerabilities, app developers should sanitize an input of filename. For example, before storing external resources coming from networks, it is important to filter out any characters that should not be included in a filename such as "./../".

In addition, recently, the CERT Division (http://www.cert.org/) updated its secure coding standards to show how to prevent arbitrary overwriting vulnerabilities using unsafe `ZipInputStream`. In the CERT Oracle Coding Standard for Java [16], with the compliant code example, the standard shows that directory traversal or path equivalence vulnerabilities can be eliminated by canonicalizing the path name and then validating the location before extraction. To prevent remote code injection attacks, developers should comply with this coding style when they need to implement ZIP archive downloads from external servers. Note that filename sanitization eliminates *CII*.

Secure Code Execution. If app developers can employ secure APIs (such as `SecureDexClassLoader` [4]), which load and execute the downloaded executables in a secure manner, attackers would not be able to execute any arbitrary code within the context of an app even when successfully injecting their payload. During secure code execution, the involved API retrieves the certificate of the developer that signed and published the given code and verifies the downloaded code, which is cryptographically signed, using the retrieved certificate. Naturally, to implement such secure APIs, all possible trigger points described in Section 4.3 should be considered. Note that secure code execution eliminates *CIII*.

Use of Secure Communication Protocol. An ideal solution for preventing remote code injection attacks is to use a secure communication protocol (such as HTTPS) to download external resources. However, applying HTTPS for all communications is virtually impossible due to performance issues and operational costs. Because DRUs (such as downloading image files) occur very frequently in Android apps nowadays, applying HTTPS for all DRUs may affect performance. Brian Jackson [36] showed that making a lot of short requests over HTTPS will be slower than HTTP. In addition, the servers that provide HTTPS involve issuing and managing a certificate. Because of this, operational costs will increase as opposed to HTTP. Furthermore, there may be vulnerabilities in SSL/TLS implementations [9, 11]. App developers should, at the very least, apply HTTPS to their sensitive communications such as downloading of executables (.dex, .so, etc.), self-updates, or other critical procedures. Note that use of secure communication protocol eliminates *CI*.

7.2. Limitations

Implicit Control Flows. Our detection tool is subject to the limitations of static flow analysis. Consequently, it does not identify all implicit control flows introduced by callbacks. This can potentially affect our program slicing component. If code snippets that download external resources are invoked from a callback that is not handled by the detection tool, we cannot locate the code snippets inside the ICFG, which is required for accurate program slicing. Although we have added extensions that support threading classes such as `AsyncTask`, `Thread`, and `Runnable`, it is not complete. Discovering Android's implicit callbacks is an active area of research, and a number of studies [37–39] are currently devoted to addressing this issue.

Dynamic URLs. Even though DRU code snippets have been shown to be vulnerable to remote code injection attacks (meaning that there are file overwrite vulnerabilities and trigger points in the app), our detection tool cannot definitively conclude whether a remote code injection attack can be accomplished by a network attacker. This is because of *dynamic URLs*. Unlike *static URLs*, which are hardcoded in apps, dynamic URLs are only present at runtime. For example, if an app requests update metadata from a server and retrieves its update URL from the received metadata, the possibility of an attacker launching an attack is contingent on the retrieved URL. That is, if the URL starts with "*https://*," the app is not vulnerable to remote code injection attacks, whereas with "*https://*" it is vulnerable. Because of the limitation of static analysis, our detection tool cannot detect this, and hence, we consider dynamic URLs to be potentially vulnerable to remote code injection attacks. Note that, for the same reason, the detection tool cannot handle

runtime libraries of dynamically loaded classes. We believe that these dynamic issues can be addressed by leveraging dynamic program analysis, as exemplified by Rocha et al. [40] and Sounthiraraj et al. [11].

8. Related Work

Code Injection Attacks in Android Apps. Poeplau et al. [6] showed that code-loading techniques are often implemented in a vulnerable manner that allows attackers to replace the legitimate codes with malicious codes. They categorized the code-loading techniques into five different groups: *class loaders*, *package context*, *native code*, *APK installation*, and *Runtime.exec*. Subsequently, they showed that attackers can abuse these techniques via insecure downloads or unprotected storage to execute their malicious codes. Their work is related to our study as it also covers remote code injection attacks, and some of the code-loading techniques they categorized can be considered as code trigger points. However, there are several differences between this work and ours: (1) They only focused on code resource (executables such as .apk, .so, and .dex) downloads, while we also focus on other resources such as ZIP archives and image files as well as executables. (2) They only checked the existence of DCL component (i.e., they did not confirm if network attacks are feasible or not), while we checked if remote code injection is feasible using automatic static data flow analysis.

OS Update Attacks. Xing et al. [41] focused on the upgrading logic in the Android platform (specifically, the Package Management Service (PMS)) and found a new type of privilege escalation vulnerability, called *Pileup*, that occurs when the user upgrades the operating system on the device. By exploiting Pileup vulnerabilities, malicious apps can silently acquire system capabilities that are valid only in the new operating system after an upgrade whereas they did not exist in the old one. Note that threat model and the target (OS) of this work are different from ours.

Script Injection Attacks in Android Apps. Several studies have been conducted on script injection attacks in Android apps [42–46]. Jin et al. [43] found a new class of script injection attacks on HTML5-based mobile applications. They identified many channels for script injections including barcode, SMS, file system, Contact, Wi-Fi, and NFC and showed that all these channels can be abused for script injection attacks such as Cross-Site Scripting (XSS). Hassanshahi et al. [42] presented web-to-app injection (W2AI) attacks, which allow malicious web attackers to inject malicious scripts by exploiting a web-to-app communication bridge, and showed that a successful attack can abuse WebView and Android native app interfaces. Zhang and Du [46] analyzed Android Clipboard and found that Clipboard data manipulation can lead to common script injection attacks, such as JavaScript injection and command injection.

Smith [45] analyzed injection attacks on Android OS and subsequently developed a detection tool based on taint analysis. The developed tool finds data flows in Android apps that lead from the input points to SQLite or OS Shell APIs. The OS Shell APIs that load and execute injected scripts are the same as one type of our trigger points; however, by contrast, our work does not rely on data flows between input points and APIs. As described in Section 4, attackers can trigger injected codes using arbitrary overwriting vulnerabilities even without abusing the inputs. Therefore, the trigger points are independent of data flows. Hence, we only identify whether the trigger points are reachable or not.

Program Analysis in Android Apps. Prior studies on Android app analysis focused on either discovering information leakage [25, 28, 47, 48] or identifying app misbehaviors [11, 26, 49, 50].

Taint analysis tracks information flows to reveal unintended information leakage. TaintDroid [47] monitors an app's behavior in real time and performs dynamic taint analysis to detect privacy-sensitive information leaks in Android. While it is more accurate than static analysis, achieving high code coverage is a significant challenge. Dynamic analysis can also be fooled by malicious apps that act benign when they recognize that they are being analyzed [47, 48].

To overcome this challenge, many studies employ static analysis [25, 26, 28, 48, 49, 51]. These studies commonly reconstruct interprocedural control-flow graphs (ICFGs) by modeling the Android app's lifecycle. By analyzing the ICFG and data dependencies, they identify whether a path exists from a source to a sink (usually network I/O APIs). In this work, we leverage FlowDroid [28], a static taint analysis tool, to reconstruct the ICFG. CryptoLint [49] detects misuse of cryptographic libraries via static program slicing. SMV-Hunter [11] identifies Android apps that fail to properly validate SSL certificates based on a combination of static and dynamic analysis.

9. Conclusion

Android apps often rely on external servers to dynamically update a variety of resources such as executables, images, and temporary files, at runtime. However, these dynamic resource updates can be vulnerable to remote code injection attacks. For example, apps that download code resources (such as .so and .jar) can be abused by network attackers attempting to replace or modify the downloading codes.

While remote code injection attacks against such code resource updates are known, attacks against other resource updates and their impact are still largely unknown. As we have shown in this paper, when an app contains file overwrite vulnerabilities in the dynamic resource update and also contains possible trigger points, attackers can still carry out remote code injection attacks by exploiting these vulnerabilities.

In this work, to identify these kinds of threats, we first investigated three conditions for successful remote code injection attacks: *no validation checks* or *bypassable validation checks*, *file overwriting vulnerabilities*, and *trigger points*. We then developed a static detection tool that automatically identifies these conditions based on heuristics, string analysis, and data dependency analysis. Finally, we applied the detection tool to a large dataset comprising 9,054 apps, consisting of

official market (Google Play), third-party market (Tencent Myapp), and preinstalled apps (system apps). Consequently, we discovered a total of 53 vulnerable apps comprising 25 official market apps and 28 third-party apps (including popular apps and libraries). Our results can provide a lower bound on the number of vulnerable apps in the wild.

Appendix

A. Program Slicing

The goal of the program slicer is to output all statements that affect network operations and to identify dependencies between slices for further analysis. After the ICFG is reconstructed, the program slicer analyzes all of the converted Jimple statements to extract the set of Jimple statements that keep the interesting program behaviors (i.e., network behaviors), such as downloading external resources. Given a variable v in program p, the output of the program slicing component consists of all statements $\{s\}$ in p that possibly affect the value of v. In other words, this implies that the program slicer identifies data dependencies between the value of v and the statement s that would exist when executing the app.

B. Dataset Collection

The first dataset consists of 4,718 diverse Android apps. To download the apps from the official Google Play marketplace, we implemented a crawler that uses the Google Play API with a device ID and a Google account. Using this crawler, we downloaded apps from 28 categories in the market. The second dataset consists of 2,967 apps collected from the Tencent Myapp market, which is the most popular Android market in China and dominates the market with a 24% market share in May 2016 [52]. We crawled the app download URL links from the Tencent Myapp website and directly downloaded .apk files from 29 categories on the site.

The third dataset is a system application set containing 1,369 apps. In accordance with recent reports [53, 54], we selected two popular smartphone manufacturers (Samsung and Huawei) and eight carriers (Verizon, AT&T, Sprint, US Cellular, T-Mobile, SK Telecom, KT, and LG U+) from two countries (USA and South Korea) to analyze their preinstalled system apps. We downloaded 148 of the latest factory images from two sources: 118 Samsung images from [32] and 30 Huawei images from [33]. To extract system apps from the images, we first decompressed each image to get a `system.img` file, which is a `/system/` directory for a running Android and contains all the system apps. Then, we converted the `Android sparse image` format to `ext4` format to mount the image. Finally, we extracted all the .apk files under the `/system/app` and the `/system/priv-app` directories. Note that, to remove duplicates, we selected the latest .apk based on time created when we ran into apps with the same name. In total, we gathered 1,369 system apps covering 46 smartphone models having Android version ranging from 4.1.2 to 6.0.1.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-2015-0-00403) supervised by the IITP (Institute for Information & Communications Technology Promotion).

References

[1] The Statistics Portal, “Global market share held by smartphone operating systems 2009-2015, by quarter,” http://www.statista.com/statistics/266136/global-market-share-heldby-smartphone-operating-systems/.

[2] “Google I/O Developer conference,” https://events.google.com/io2016/.

[3] AppBrain, “Number of Android applications on Google Play,” http://www.appbrain.com/stats/number-of-android-apps/.

[4] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi, “Grab ’n run: Secure and practical dynamic code loading for android applications,” in *Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC 2015*, pp. 201–210, USA, December 2015.

[5] AppBrain, “Android Ad networks,” http://www.appbrain.com/stats/libraries/ad/.

[6] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications,” in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA.

[7] B. Watson, “Your Browsers Are Belong To Us,” *In INFILTRATE*, April 2016.

[8] R. Welton, “Remotely Abusing Android,” in *Black Hat London 2015*, 2015.

[9] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, “Why Eve and Mallory love Android: An analysis of Android SSL (in)security,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012*, pp. 50–61, USA, October 2012.

[10] Y. Jia, Y. Chen, X. Dong, P. Saxena, J. Mao, and Z. Liang, “Man-in-the-browser-cache: persisting HTTPS attacks via browser cache poisoning,” *Computers & Security*, vol. 55, no. 1, pp. 62–80, 2015.

[11] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, “SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps,” in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA.

[12] “Google Play Developer Policy Center,” https://play.google.com/about/privacy-andsecurity.html,#malicious-behavior.

[13] Android Developers, “Android Resource Types,” http://developer.android.com/intl/en/guide/topics/resources/available-resources.html.

[14] B. Alotaibi and K. Elleithy, “Rogue Access Point Detection: Taxonomy, Challenges, and Future Directions,” *Wireless Personal Communications*, vol. 90, no. 3, pp. 1261–1290, 2016.

[15] "Appnext - App Monetization Platform," http://www.appnext.com/.

[16] IDS04-J, "Safely extract files from ZipInputStream," https://www.securecoding.cert.org/confluence/display/java/IDS04-J.+Safely+extract+files+from+ZipInputStream.

[17] A Developers, "Building Apps with Over 64K Methods," http://developer.android.com/tools/building/multidex.html.

[18] "Runtastic: Running, Cycling & Fitness GPS Tracker," https://www.runtastic.com/.

[19] "Umeng PushSDK - Push Notifications Service," http://push.umeng.com/.

[20] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *CETUS*, and L. Hendren. The Soot framework for Java program analysis, a retrospective. In CETUS, 2011.

[21] A. Bartel, J. Klein, and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP 2012*, pp. 27–38, China, June 2012.

[22] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[23] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.

[24] D. Geneiatakis, I. N. Fovino, I. Kounelis, and P. Stirparo, "A Permission verification approach for android mobile applications," *Computers & Security*, vol. 49, pp. 192–205, 2015.

[25] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," *In NDSS*, vol. volume 14, p. page, 2012.

[26] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012*, pp. 229–240, USA, October 2012.

[27] 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, pp. 2338–2349, 2015.

[28] S. Arzt, S. Rasthofer, C. Fritz et al., "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pp. 259–269, ACM, June 2014.

[29] H. Choi, J. Kim, H. Hong, Y. Kim, J. Lee, and D. Han, "Extractocol," *Computer Communication Review*, vol. 45, no. 5, pp. 593-594, 2015.

[30] J. Kim, H. Choi, H. Namkung et al., "Enabling Automatic Protocol Behavior Analysis for Android Applications," in *Proceedings of the 12th ACM Conference on Emerging Networking Experiments and Technologies, ACM CoNEXT 2016*, pp. 281–295, USA, December 2016.

[31] "Tencent Myapp," http://android.myapp.com/.

[32] "Samsung Updates," http://samsungupdates.com/.

[33] "Huawei Consumer," http://consumer.huawei.com/.

[34] Android Developers, "Normal and Dangerous Permissions," https://developerandroid.com/guide/topics/security/permissions.html#normal-dangerous.

[35] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, Calif, USA, May 2012.

[36] B. Jackson, "Analyzing HTTPS Performance Overhead," https://www.keycdn.com/blog/https-performance-overhead/.

[37] O. Bastani, S. Anand, and A. Aiken, "Interactively verifying absence of explicit information flows in Android apps," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 299–315, 2015.

[38] Y. Cao, Y. Fratantonio, A. Bianchi et al., "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA.

[39] D. Octeau, P. McDaniel, S. Jha et al., "Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis," in *USENIX Security*, and Y. Le Traon. Effective Inter-Component Communication Mapping in Android, An Essential Step Towards Holistic Security Analysis. In USENIX Security, 2013.

[40] B. P. S. Rocha, M. Conti, S. Etalle, and B. Crispo, "Hybrid static-runtime information flow and declassification enforcement," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 8, pp. 1294–1305, 2013.

[41] L. Xing, X. Pan, R. Wang, K. Yuan, and X. F. Wang, "Upgrading your Android, elevating my malware: privilege escalation through mobile OS updating," in *Proceedings of the 35th IEEE Symposium on Security and Privacy, SP 2014*, pp. 393–408, San Jose, Calif, USA, May 2014.

[42] B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang, "Web-to-application injection attacks on android: Characterization and detection," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 9327, pp. 577–598, 2015.

[43] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: characterization, detection and mitigation," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp. 66–77, ACM, November 2014.

[44] J. Mao, R. Wang, Y. Chen, and Y. Jia, "Detecting injected behaviors in HTML5-based Android applications," *Journal of High Speed Networks*, vol. 22, no. 1, pp. 15–34, 2016.

[45] G. J. Smith, "Analysis and Prevention of Code-Injection Attacks on Android OS".

[46] X. Zhang and W. Du, "Attacks on Android Clipboard," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 8550 of *Lecture Notes in Computer Science*, pp. 72–91, Springer International Publishing, Cham, 2014.

[47] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, article 5, 2014.

[48] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 7344, pp. 291–307, 2012.

[49] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proceedings of the ACM SIGSAC Conference on*

Computer & Communications Security (CCS '13), pp. 73–83, ACM, Berlin, Germany, November 2013.

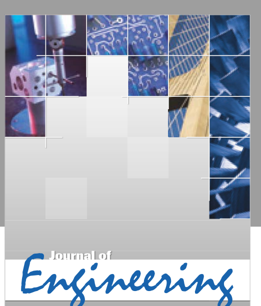
[50] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pp. 1036–1046, India, June 2014.

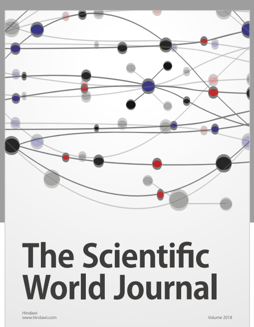
[51] K. O. Elish, X. Shu, D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Computers & Security*, vol. 49, pp. 255–273, 2015.

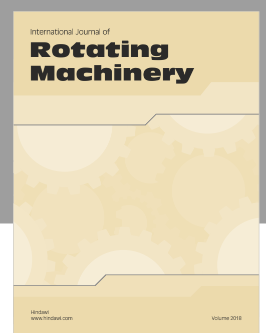
[52] Newzoo, "Top 10 Android App Stores in China," https://newzoo.com/insights/rankings/top-10-android-appstores-china/.

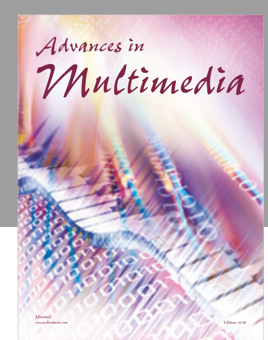
[53] Gartner, "Market Share Alert: Preliminary, Mobile Phones, Worldwide, 1Q16".

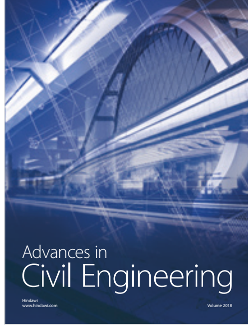
[54] Statista, "Market share of wireless subscriptions held by carriers in the U.S. from 1st quarter 2011 to 1st quarter 2016," http://www.statista.com/statistics/199359/market-shareof-wireless-carriers-in-the-us-bysubscriptions/.











Submit your manuscripts at
www.hindawi.com