sustainability

MDPI

*Article*

# Vulnerability Evaluation Method through Correlation Analysis of Android Applications

**Cheolmin Yeom and Yoojae Won ***

Department of Computer Science and Engineering, Chungnam National University, Daejeon 34134, Korea; cjfals18@cnu.ac.kr

* Correspondence: yjwon@cnu.ac.kr; Tel.: +82-42-821-6294

Received: 27 September 2019; Accepted: 21 November 2019; Published: 24 November 2019


**Abstract:** Due to people in companies use mobile devices to access corporate data, attackers targeting corporate data use vulnerabilities in mobile devices. Most vulnerabilities in applications are caused by the carelessness of developers, and confused deputy attacks and data leak attacks using inter-application vulnerabilities are possible. These vulnerabilities are difficult to find through the single-application diagnostic tool that is currently being studied. This paper proposes a process to automate the decompilation of all the applications on a user's mobile device and a mechanism to find inter-application vulnerabilities. The mechanism generates a list and matrix, detailing the vulnerabilities in the mobile device. The proposed mechanism is validated through an experiment on an actual mobile device with four installed applications, and the results show that the mechanism can accurately capture all application risks as well as inter-application risks. Through this mechanism, users can expect to find the risks in their mobile devices in advance and prevent damage.

**Keywords:** android security; android permission; inter-application vulnerability; vulnerability diagnosis

## 1. Introduction

Currently, over 76% of the world's population uses smartphones, and mobile devices offer not only phone, mail, and camera functions, but also games, mobile payments, and wallets [1,2]. Applications that enable these features can be easily downloaded and installed from application stores [3]. Because of this convenience, mobile devices and applications are being used as a key means of accessing business information in enterprises. However, as mobile devices offer more functions, they store not only phone numbers, photos, and videos, but also sensitive information such as personal information, secret keys, biometric data, and corporate data [4]. People are using mobile devices to access personal or corporate data on a regular basis, increasing the security threat to user and corporate information [5]. Consequently, the importance of data security in mobile devices is steadily increasing.

There are many operating systems (OSs) for mobile devices, including Android, iOS, and BlackBerry OS, but over 74% of mobile devices use the Android OS [6]. Android has become a target for hackers because of its high adoption among mobile devices as well as the large amount of data handled by Android. Android has features to protect user data, developer apps, devices, and networks. However, security depends on the developer's ability [7]. Some developers spend a large amount of time in building user interfaces and features but do not focus on the need for application security. The resulting vulnerabilities could inadvertently allow attackers to access sensitive data [8]. For example, Android allows applications to communicate with each other using intents for flexible operation. If a restaurant search application requests an invocation message or calls a map application to determine the location of a restaurant, a malicious attacker could intercept these messages to view and alter the contents or send malicious messages [9]. Inter-component communication (ICC) proceeds with the above intent message. If this message is not carefully controlled, it may be attacked through

methods such as intent spoofing, unauthorized intent receipt, and privilege escalation [7]. However, even with care, it is difficult to eliminate risk completely [10]. Therefore, a mechanism or tool is needed to detect vulnerabilities in mobile devices.

Currently, studies on the analysis of applications are underway. However, research on the analysis of existing applications usually proceeds by inspecting a single application and does not detect inter-application vulnerabilities. In the case of research providing analysis for multiple applications, application groups are selected arbitrarily. The user's mobile device does not know what vulnerabilities exist among applications. In this study, we propose an algorithm that decompiles all applications existing in a real user's device and analyzes the results obtained through decompilation by using detection algorithms. When the detection mechanism is terminated, it shows the risk level among all the applications and the component with vulnerable elements among the applications in the user's device.

## 2. Materials and Methods

### *2.1. Background*

#### 2.1.1. Android Architecture

Android is designed based on the Linux kernel, and its structure is shown in Figure 1. The Android architecture consists of a Linux kernel layer, hardware abstraction layer (HAL), native C/C++ libraries layer, Java Application Programming Interface (API) framework layer, and system apps layer. The Linux kernel layer is the foundation of the Android platform and leverages basic features such as threading and memory management, which includes Android runtime (ART). The Linux kernel also allows Android to take advantage of key security features. HAL provides a high-level Java API framework with a standard interface that exposes hardware features such as cameras or Bluetooth. ART allows each app to run as its own ART instance within its own process. ART provides compilation, optimized garbage collection, exception and crash reporting, and debugging to monitor specific fields. Many system components and services require native libraries written in C and C++. The native C/C++ library layer provides the libraries and grants access to the native platform library for apps that require C or C++ code. The Java API framework simplifies the core modular system configuration requirements and service recycling to form the building blocks needed to develop Android apps. System apps also work as apps for users and provide key features that developers can access in their apps [11–13].

#### 2.1.2. Android Sandboxing

Android separates applications running through sandboxing, and each application has a unique identifier. A developer can set their application's own file or another application's file as readable, writeable, or executable by explicitly displaying the application's access rights to the file owned by the application, along with the application's identifier in the system file owned by "system" or "root" [14,15].

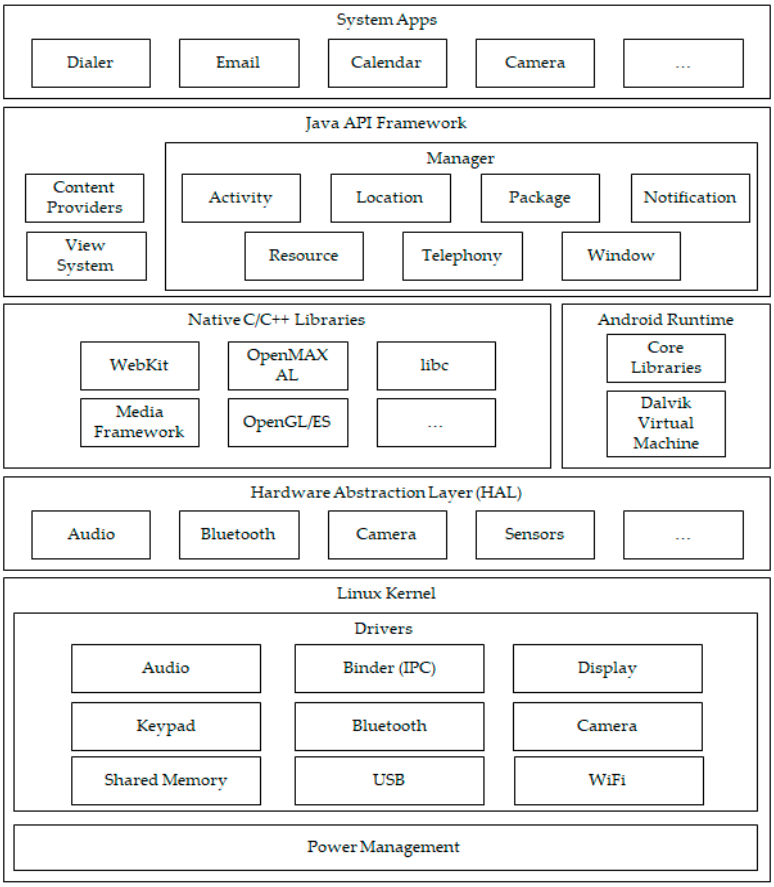


**Figure 1.** Android architecture.

### 2.1.3. Component

Android consists of four components: activity, service, content provider, and broadcast receiver. An activity provides a user interface (UI) through the display and manages the UI by performing tasks such as interaction with the user to support various types of activities performed in the application framework. An application usually consists of several activities. A service performs background processing and does not provide a UI. Services can perform various functions such as audio and file downloads while other applications are running. The content provider connects the application layer with the data layer and provides a service or necessary function to other components as a database that can be addressed by uniform resource locator (URL) in the application. The broadcast receiver receives intents from the Android application framework. Most of the broadcast messages are generated by the system. Intent messages are used by applications to request functions from other services or activities [9,16].

### 2.1.4. Android PacKage (APK) File

Figure 2 shows the process of creating an APK file. It is a Dalvik EXcute (DEX) file created using Java codes, the AndroidManifest file, and the files used in the application and is packaged in a compressed file format.

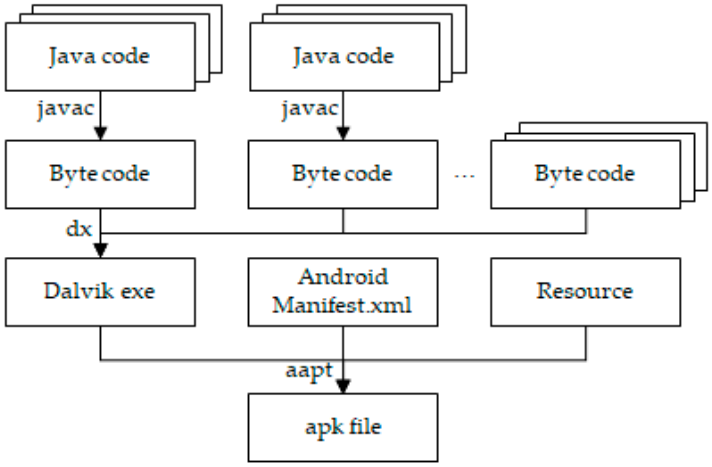


**Figure 2.** Android PacKage (APK) file creation process.

#### 2.1.5. Decompile Tools

Table 1 lists the tools for extracting and analyzing APK files. The Android debug bridge (ADB) supports the use of the Unix shell on Android devices, and the user can use ADB to run commands on emulators or real Android devices. APKtool can compile the APK file and vice versa to obtain the resulting smali file and AndroidManifest.xml file. dex2jar converts dex files into Java archive (JAR) files, and JAR files combine several Java class files and resources used by the classes into a single file for distribution. Java decompiler (JDA) is a tool that converts class files to JAVA files. Android asset packaging tool (Aapt) is a part of the Android build tool, and it provides various information from APK files.

**Table 1.** List of decompile tools.

| Name of the Tool | Function of the Tool | Output |
|---|---|---|
| Android debug bridge (ADB) | Enables Unix shell on Android | APK file |
| Android PacKage tool (APKtool) | APK decompile | AndroidManifest |
| Dalvik EXcute to Java archive (dex2jar) | DEX file decompile | JAR file |
| jad | Class file to Java file | Java file |
| aapt | Extract string values from APK file | String values |

### *2.2. Related Studies*

Applications developed by third parties in the market can be installed on Android devices. Every Android application runs in a sandbox and is given a unique user identifier (UID) and group identifier (GID). However, if an application needs to use resources or data outside its own sandbox, the app must request the appropriate permissions, which are specified in AndroidManifest. If any of these permissions involve data or resources that may contain personal information, require access to stored data, or affect the operation of other apps, the user can install the application with varying levels of trust by classifying them as dangerous. If an application has more permission than necessary, it can easily steal sensitive information.

Table 2 lists examples of permissions to suspect malicious behavior when allowed together by application. If a malicious photo application has location information, permission to read short message service (SMS)s, and access to the Internet, the application can potentially read the user's location or SMSs and send the information to a malicious party via the Internet. In order to prevent such attacks, a check must be performed for the combination of risk authorities [11,17].

**Table 2.** Permissions to suspect malicious behavior when allowed together.

| Malicious Behaviors | Permissions |
| --- | --- |
| Malicious fee-deduction | RECEVE_MSM<br>RECEVE_MMS<br>SEND_MSM<br>SEND_MMS<br>READ_SMS<br>CALL_PHONE<br>CALL_PRIVILEGED |
| Remote control | RECEVE_MSM<br>RECEVE_MMS<br>SEND_MSM<br>SEND_MMS<br>READ_SMS<br>INTERNET<br>ACCESS_NETWORK_STATE<br>CHANGE_NETWORK_STATE<br>ACCESS_WIFI_STATE<br>CHANGE_WIFI_STATE |
| Information theft | READ_CONTACTS<br>ACCESS_FIND_LOCATION<br>ACCESS_COARSE_LOCATION<br>READ_CALL_LOG<br>WRITE_CALL_LOG<br>READ_PHONE_STATE<br>INTERNET<br>ACCESS_NETWORK_STATE<br>CHANGE_NETWORK_STATE<br>ACCESS_WIFI_STATE<br>CHANGE_WIFI_STATE |
| Fee consumption | INTERNET<br>ACCESS_NETWORK_STATE<br>CHANGE_NETWORK_STATE<br>ACCESS_WIFI_STATE<br>CHANGE_WIFI_STATE |
| Rogue behavior | INTERNET<br>ACCESS_NETWORK_STATE<br>CHANGE_NETWORK_STATE<br>ACCESS_WIFI_STATE<br>CHANGE_WIFI_STATE<br>INSTALL_PACKAGE<br>DELETE_PACKAGE |

Android applications are separated from the sandbox and run independently on the principle that they do not trust each other. Component calls using intents can be explicitly or implicitly communicating within one application or externally with other applications. This could result in the elevation of privilege vulnerabilities that invoke and use components of external applications with higher permission [18]. The above vulnerability occurs because all applications gain access when the exported attribute of the provider tag in the AndroidManifest file is set to true or unspecified. The method in Table 3 is used for activity, content provider, and broadcast receive among Android's components [3,9].

**Table 3.** List of methods to send information over the Intent.

| Usage | Method |
| --- | --- |
| To Receiver | sendBroadcast (Intent i)<br>sendBroadcast (Intent i, String recvrPermission)<br>sendOrderedBroadcast (Intent i, String recvrPermission)<br>sendOrderedBroadcast (Intent i, String recvrPermission, BroadcastReceiver receiver, . . . )<br>sendStickyBroadcast (Intent i)<br>sendStickOderedBroadcast(Intent i, BroadcastReceiver receiver, . . . ) |
| To Activity | startActivity (Intent i)<br>startActivityForResult (Intent I, int requestCode) |
| To Service | startService (Intent i)<br>bindService (Intent i, ServiceConnection conn, int flags) |

For application analysis, the Java source can be obtained using various tools from the APK file. APKtool can be used to analyze the Android application binaries and unpackage the packaged APK. Furthermore, a DEX file can be obtained by decompressing the APK file, which is in the Dalvik virtual system format and can be converted to a class file by using a tool called dex2jar. Finally, class files can be decompiled into JAVA files by using a tool called jd-gui [19].

### *2.3. Diagnostic Mechanism*

The diagnostic mechanism identifies and lists the risks in an Android application and uses them to determine inter-application risks. To apply this mechanism, we need to analyze the AndroidMenifest file containing the configuration of the application, the JAVA source file of the application, and finally the strings used by the application [20]. AndroidMenifest files and JAVA source files can be obtained by decompilation, and string values can be extracted from the APK file. Automated decompilation can be used to obtain information on all applications, and detection algorithms can analyze the results of decompilation and identify applications that could be attacked by malicious applications.

#### 2.3.1. Automated Decompiling

Figure 3 shows how an application installed on a mobile device is automatically decompiled. After connecting the mobile device with the "Universal Serial Bus (USB) debug" option enabled to a computer and running the automated decompiling, the program first uses ADB to find the installation file paths of all the applications installed on the phone, extracts the APK file, and creates a directory with each application's name. Second, APKtool is used to obtain AndroidManifest from each APK file. Third, string values are obtained using the aapt tool, which extracts string values from the APK file. Fourth, the APK file's extension is changed to zip, the archive is extracted to obtain the corresponding DEX file, and the corresponding JAR file is obtained through dex2jar. Fourth, the JAR file is extracted to obtain class files, which are divided into several directories and extracted. The paths of these class files are found, and the class files are finally obtained using jad.

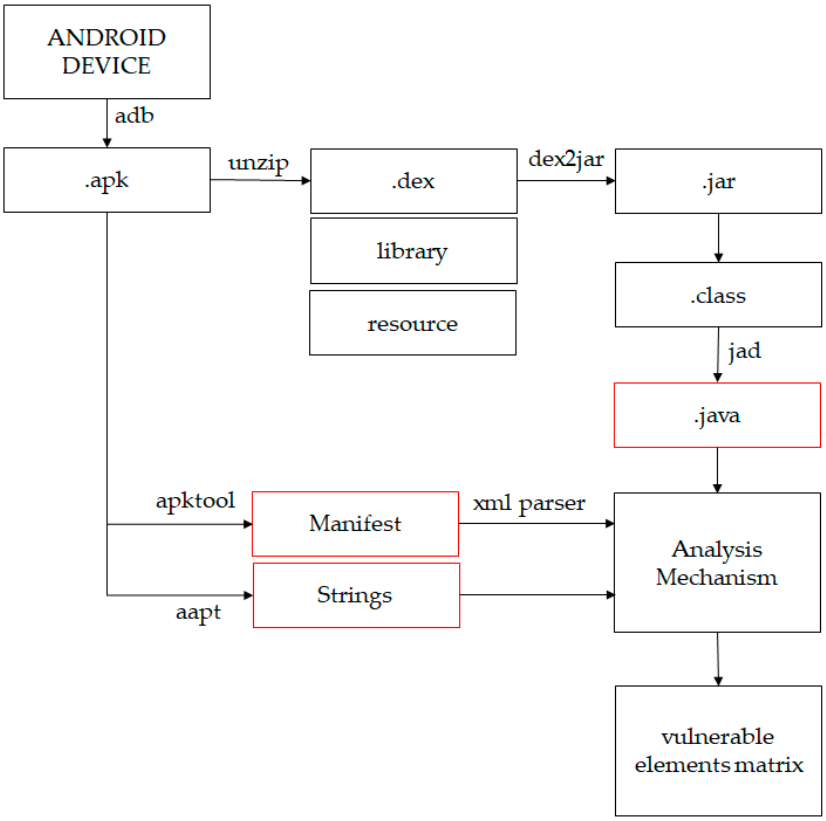


**Figure 3.** Application decompile order and output for each step.

### 2.3.2. Detection Algorithm

Figure 4 shows the flow of the detection algorithm. Through the AndroidMenifest file in extensible markup language (XML) format obtained through decompilation before starting the algorithm, the component's information such as permission as well as the activity and provider included in the application can be obtained. Furthermore, functions that can send intent or query messages through the Java source code can be found. When the algorithm starts, it initializes the caution level, which indicates the degree of danger. It then starts checking for permissions, attributes, source code, and string values that result from decompilation.

In this case, an application that can show malicious behavior is referred to as a subject application, and an application that can be damaged by it is referred to as a target application. First, a string value suspected to be a key values or hash values are output by checking the length of the string value of the target application. Hardcoded key values or hash values can be leaked and cause more damage. Second, the subject application's permissions are examined, and the caution level is increased to 1 if based on using many of the privileges specified in Table 2 or requesting too many privileges, even if not specified in Table 2. Third, the existence of a function that can send an intent message or query message based on Table 3 in the Java source code of the subject application is checked. If such a function exists, it is checked whether the exported property in the target application's tag is true or unspecified so that the message can be received. If possible, the caution level is increased by 1. Finally, the message can be received because the exported property in the target application's tag is true or unspecified. If authority values exist and if the same value exists among the string values of the subject application, the data can be accessed.

When the above mechanism is terminated, the risk level between all combinations of subject application and target application is classified as Caution for 1, Warning for 2, and Danger for 3 according to the caution level.

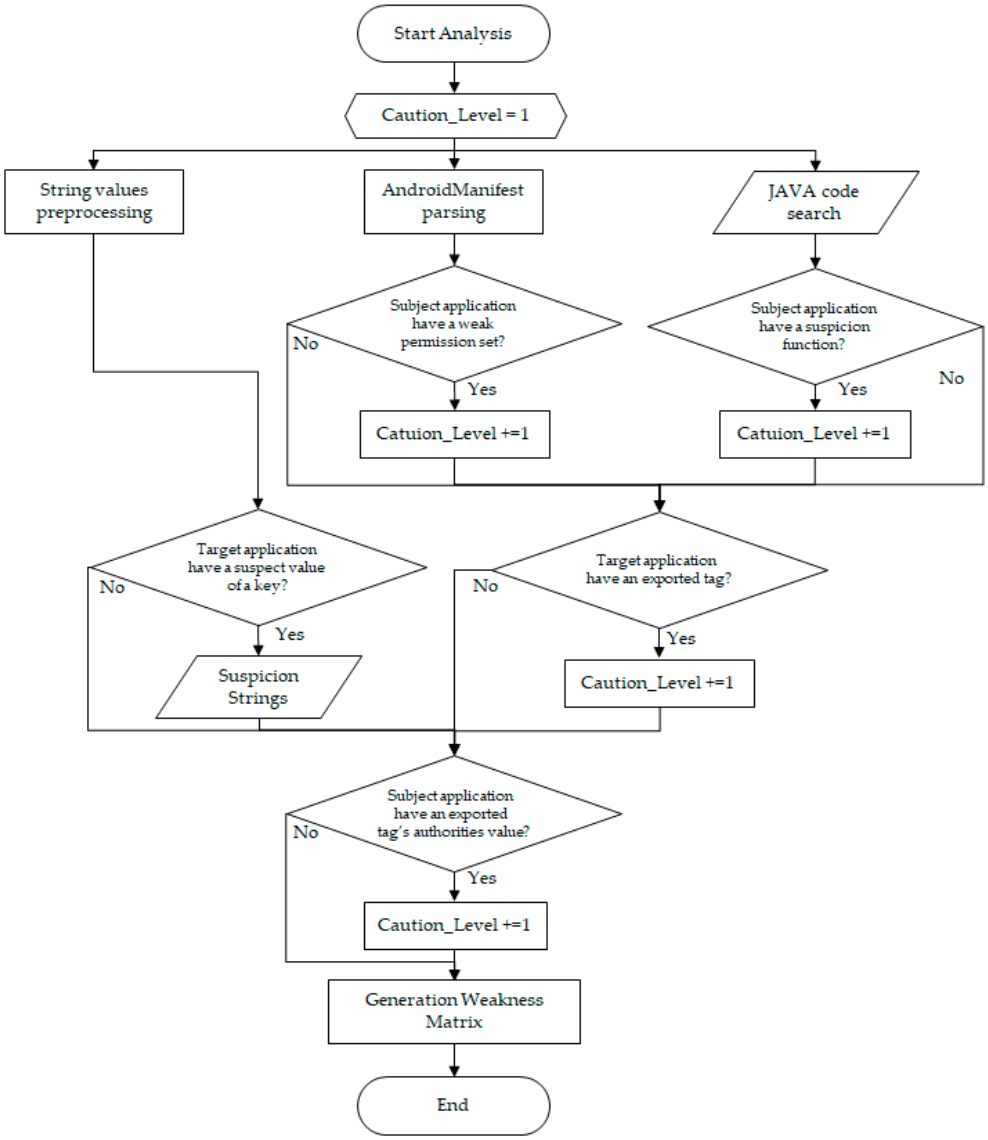


**Figure 4.** Flowchart for vulnerable application detection algorithm.

## 2.4. Experiment

The proposed mechanism detects vulnerable elements among the applications installed in Android and implements the mechanism as a program. The results of the program are divided into list mode, which shows the values and details of detected items, and matrix mode, which shows the degree of risk among all applications.

### Experiment Environment

Four applications were implemented for the experimentation of the mechanism. An application can use dangerous permission combinations, an abnormal number of permissions, components that can send messages, components that have an exported attribute of true or unspecified, intent or query messages, and authority values to externally accessible providers to ensure that the elements checked by the mechanism work properly. Intentionally, Subject_Application is implemented to be dangerous to Target_Application.

Table 4 shows the characteristics of each of the four applications. Among them, the application is designed to cause a problem between Target_Application and Subject_Application.

**Table 4.** Applications in the experimental environment.

| | Subject_Application | Smart Messenger | Emergency Call | Target_Application |
|---|---|---|---|---|
| Have dangerous permission sets? | O | X | X | X |
| Have too many permissions? | X | O | X | X |
| Have an exported provider? | X | O | X | O |
| Have suspicion functions? | O | X | O | O |
| Use authority values? | O | X | X | O |

Figure 5 shows the Target_Application's AndroidManifest. Target_Application plays the role of the application under attack. It has one exported property each with true, false, and unspecified provider components, as well as authority values to access each provider.

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="Target_Application"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:ignore="GoogleAppIndexingWarning">
    <activity
        android:name=".DataActivity"
        android:label="DataActivity"
        android:theme="@style/AppTheme.NoActionBar"/>
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <provider
        android:name=".ScheduleList"
        android:authorities="com.fury.Target_Application.ScheduleList"
        android:exported="false" />
    <provider
        android:name=".SensitiveData"
        android:authorities="com.fury.Target_Application.SensitiveData"
        android:exported="true" />
    <provider
        android:name=".PrivacyData"
        android:authorities="com.fury.Target_Application.PrivacyData"/>
</application>

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

**Figure 5.** Target_Application's AndroidManifest.xml.

Figure 6 shows the Subject_Application's AndroidManifest. Subject_Application plays the role of the attacking application, and it is designed to have dangerous combinations of permissions, functions to send intent messages, and authority values of the content provider that Target_Application possesses

```
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.ACCESS_FIND_LOCATION"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>

<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="Subject_Application"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:ignore="GoogleAppIndexingWarning">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

**Figure 6.** Subject_Application's AndroidManifest.xml.2.4.2. Experiment Method.

The program then runs after connecting the mobile device to the computer. If the list mode is selected after the program is run, the program will have the permissions of each application, a string value of a certain length, a component with the exported attribute set to true or unspecified, and the authority values of the provider. Create a TXT file and list the functions that can send messages. If the matrix mode is selected, the detection algorithm introduced in Section 2.3.2 is used to output the inter-application risks as a comma-separated values (CSV) file.

After a mechanism experiment with a custom application, the device was installed with an application ranking up to 8th in the app market, including the PlayStore, and the mechanism was redone. The eight applications are WhatsApp, Messenger, TikTok, Facebook, Instagram, SHAREit, YouTube, and LIKE Video [21]. Among them, WhatsApp transformed to access content provider using Instagram's authentication value on APK file before installation.

## 3. Results

The results of the program are divided into the list mode, which shows the values and details of detected items, and matrix mode, which shows the degree of risk among all applications.

### *3.1. List Mode*

When the list mode is selected, the program displays a screen showing the decompilation of all applications installed on the device and checks on the elements, and the elements to check are output.

Figure 7 shows the output for permission and string check. First, we print the list of permissions, and we can see which applications have dangerous permission combinations and are requesting too many permissions. Second, the string suspected of being a hard-coded key value or hash value can be printed out according to the length.

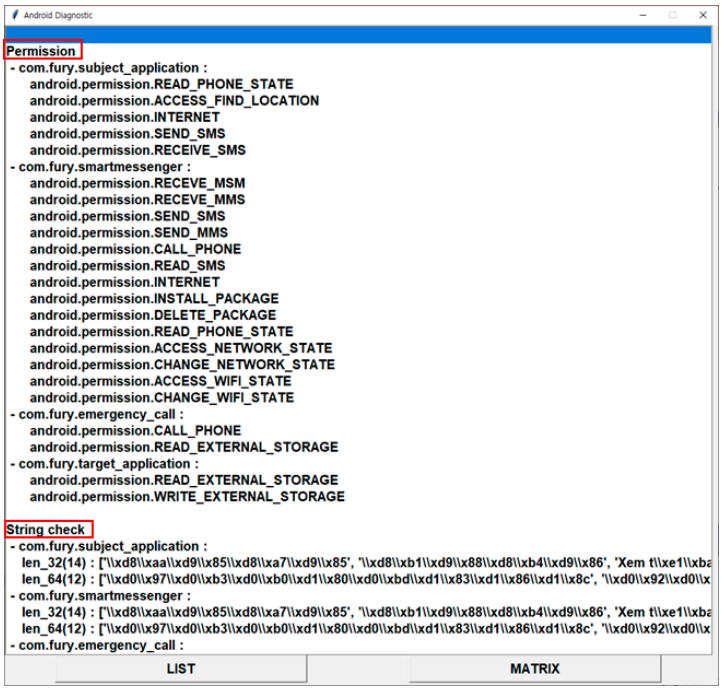


**Figure 7.** List mode's output (permission and string check).

Figure 8 shows the output for the element state, authority values, and suspicious functions. The third element state, following Figure 6, shows the tags and names of the externally accessible items of every application element. Fourth, the authority values, if any, of the externally accessible providers are printed. Fifth, the functions that can send intent or query messages to themselves or other applications are printed as suspicious functions.

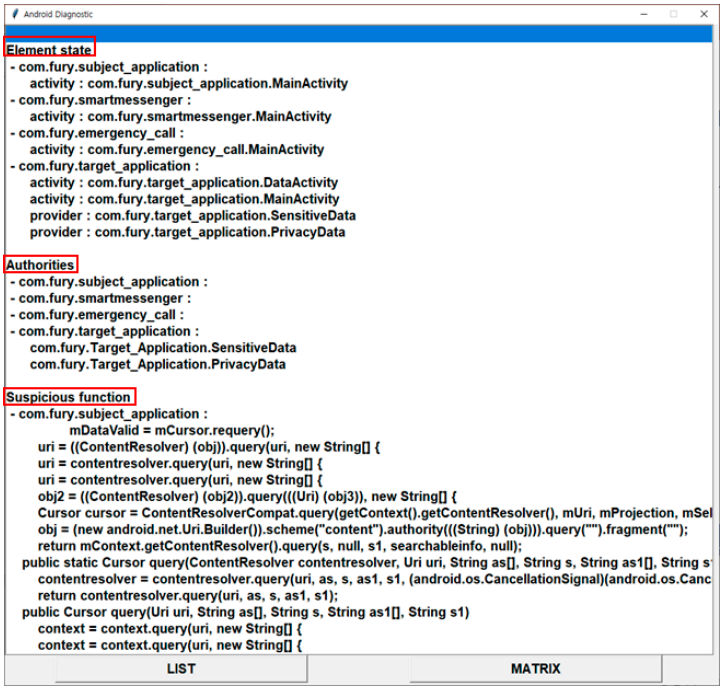


**Figure 8.** List mode's output (element state, authority values, and suspicious functions).

### 3.2. Matrix Mode

The matrix mode, like list mode, decompiles all applications installed on the device and examines the elements. Subsequently, the risks in the analysis are measured and numerically analyzed for the effects each element has on each other.

Table 5 lists the results obtained in the CSV file when the matrix mode is selected. The table lists the names of all the applications on the top and left. At the top, it is assumed that the application in the row is malicious. If the application has too many privileges or a dangerous combination of privileges, an asterisk is placed in front of the application name. When it is on the left, assuming that the application in the column is attacked, the field at which the two applications intersect is divided into Caution, Warning, and Danger according to caution level.

**Table 5.** Matrix Mode output.

| | *Subject_Application | *Smartmessenger | Emergency Call | Target_Application |
|---|---|---|---|---|
| **subject_application** | Caution | Caution | - | - |
| **smartmessenger** | Caution | Caution | - | - |
| **emergencycall** | Caution | Caution | - | - |
| **target_application** | Danger | Warning | Caution | Caution |

### 3.3. App Market Application's Result

On the mobile device, the top eight applications in the App Market download ranks have been installed and the diagnostic mechanism has been performed. The installed application had a number of exported providers and their authorities, with three results from VideoLike, two from WhatsApp, seven from Instagram, two from SHAREit, and five from TikTok. Table 6 shows the matrix obtained as a result of the execution and shows the caution between each application. Among the eight programs, we couldn't find any results sharing the authorities, but we can see the warning because we are sending and receiving messages. In particular, because WhatsApp has access to Instagram's content provider, it is marked Danger in the Instagram column of that row.

**Table 6.** App Market Application's result.

| | *YouTube | *VideoLike | *WhatsApp | Instagram | *SHAREit | *TikTok | *Facebook | *Messenger |
|---|---|---|---|---|---|---|---|---|
| **YouTube** | Caution | Caution | Caution | - | Caution | Caution | Caution | Caution |
| **VideoLike** | Warning | Warning | Warning | Caution | Warning | Warning | Warning | Warning |
| **WhatsApp** | Warning | Warning | Warning | Caution | Warning | Warning | Warning | Warning |
| **Instagram** | Warning | Warning | Danger | Caution | Warning | Warning | Warning | Warning |
| **SHAREit** | Warning | Warning | Warning | Caution | Warning | Warning | Warning | Warning |
| **TikTok** | Warning | Warning | Warning | Caution | Warning | Warning | Warning | Warning |
| **Facebook** | Caution | Caution | Caution | - | Caution | Caution | Caution | Caution |
| **Messenger** | Caution | Caution | Caution | - | Caution | Caution | Caution | Caution |

## 4. Discussion

In order to confirm the proposed mechanism, experiments were conducted on Android with four applications containing vulnerable elements and a PC capable of executing the risk detection mechanism. As a result, the TXT file with the same contents as the result screen for checking the list of vulnerable elements was obtained through List Mode, and the CVS file containing the matrix representing the risk between applications was obtained through Matrix Mode.

As a result of the experiment, Figure 6 shows the permissions of each application. SmartMessenger application has 14 permissions. Subject_Application has five permissions, but it has a combination of permissions that can be dangerous. Consequently, SmartMessenger and Subject_Application are classified as dangerous applications, with an asterisk in front of their names at the top of Table 5. Moreover, applications that are targeted by these two applications receive a caution. Figure 4 shows that some detected activities are not supposed to have the exported attribute set. Among

them, Target_Application has three providers. SmartMessenger and Subject_Application, on the other hand, both have a suspicious function that can send intent or query messages. Accordingly, Target_Application receives one more caution as it could receive indiscriminate messages from SmartMessenger, Subject_Application, and Emergency_Call, respectively. Finally, according to the authority values in Figure 7, there exists an authority value among the properties of the accessible provider of Target_Application that was created for use by Subject_Application, as described in Section 2.4. Therefore, Subject_Application receives one more caution because it can access the data of Target_Application through the content provider. In Table 5, Target_Application can receive indiscriminate messages from Emergency_Call and receives one caution. Furthermore, it can receive indiscriminate messages from SmartMessenger, which is classified as a dangerous application. Finally, when Target_Application is targeted by Subject_Application, it receives three cautions and is classified as Danger because it can receive indiscriminate messages from Subject_Application, which is classified as a dangerous application and can possibly access data through authority values.

Experiments show that the risk detection mechanism works correctly. In 2019, an average of 80 applications are installed on a user's mobile device [22]. It is difficult to diagnose dozens of applications with existing application vulnerability diagnosis studies. However, the proposed mechanism automatically checks dozens of applications and presents a matrix of risks among all applications. The mobile device user can prevent the risk of the mobile device by resetting the permissions granted to the application or removing the application through the result of the mechanism.

## 5. Conclusions

Existing studies have not been able to detect security risks outside an application because they either inspected only a single application or examined multiple applications with unclear criteria for selecting an application group. The mechanism proposed in this study not only finds the risks that exist in all applications installed on a user's device, but also finds all the inter-application risks. We also visualized the inter-application risks in a matrix. These results can be used to identify inter-application risks and prevent attacks or data leakage and corruption resulting from the elevation of privilege. However, at present, this mechanism does not check for the latest vulnerabilities, such as those in the download provider. Future research will be aimed at upgrading the mechanism such that more elements will be detectable.

**Author Contributions:** C.Y. designed the research framework, analyzed the data, and wrote the paper. Y.W. guided this work and provided extensive revisions during the study. Both authors have read and approved the final manuscript.

**Funding:** Short Message Service.

**Acknowledgments:** This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2019-2016-0-00304) supervised by the IITP (Institute for Information and communications Technology Planning and Evaluation).

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Sung, Y.; Park, J.H. Future trends of blockchain and crypto currency: Challenges, opportunities, and solutions. *J. Inf. Process. Syst.* **2019**, *15*, 457–463.
2. Kang, J. Mobile payment in Fintech environment: Trends, security challenges, and services. *Hum.-Cent. Comput. Inf. Sci.* **2018**, *8*, 32. [CrossRef]
3. Jiang, Y.Z.X.; Xuxian, Z. Detecting Passive Content Leaks and Pollution in Android Applications. In Proceedings of the 20th Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23 April 2013.
4. Perez, A.J.; Zeadally, S.; Jabeur, N. Security and privacy in ubiquitous sensor networks. *J. Inf. Process. Syst.* **2018**, *14*, 286–308.

5. Fang, Z.; Han, W.; Li, Y. Permission based Android security: Issues and countermeasures. *Comput. Secur.* **2014**, *43*, 205–218. [CrossRef]
6. Smartphone Ownership Is Growing Rapidly Around the World, but Not Always Equally, Pew Research Center. Available online: https://www.pewresearch.org/global/2019/02/05/smartphone-ownership-is-growing-rapidly-around-the-world-but-not-always-equally/ (accessed on 25 May 2019).
7. Faruki, P.; Bharmal, A.; Laxmi, V.; Ganmoor, V.; Gaur, M.S.; Conti, M.; Rajarajan, M. Android security: A survey of issues, malware penetration, and defenses. *IEEE Commun. Surv. Tutor.* **2014**, *17*, 998–1022. [CrossRef]
8. Demissie, B.F.; Ghio, D.; Ceccato, M.; Avancini, A. Identifying Android Inter App Communication Vulnerabilities Using Static and Dynamic Analysis. In Proceedings of the International Conference on Mobile Software Engineering and Systems, Austin, TX, USA, 16–17 May 2016; pp. 255–266.
9. Chin, E.; Felt, A.P.; Greenwood, K.; Wagner, D. Analyzing Inter-application Communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, MD, USA, 28 June–1 July 2011; pp. 239–252.
10. Xie, J.; Fu, X.; Du, X.; Luo, B.; Guizani, M. Autopatchdroid: A Framework for Patching Inter-app Vulnerabilities in Android Application. In Proceedings of the 2017 IEEE International Conference on Communications 2017, ICC, Paris, France, 21–23 May 2017; pp. 1–6.
11. Singh, P.; Tiwari, P.; Singh, S. Analysis of malicious behavior of android apps. *Procedia Comput. Sci.* **2016**, *79*, 215–220. [CrossRef]
12. Nauman, M.; Khan, S.; Zhang, X. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security 2010, Beijing, China, 13–16 April 2010; pp. 328–332.
13. Platform Architecture. Available online: https://developer.android.com/guide/platform (accessed on 20 September 2019).
14. Davi, L.; Dmitrienko, A.; Sadeghi, A.R.; Winandy, M. Privilege Escalation Attacks on Android. In Proceedings of the International Conference on Information Security, Boca Raton, FL, USA, 25–28 October 2010; pp. 346–360.
15. Sato, R.; Chiba, D.; Goto, S. Detecting Android Malware by Analyzing Manifest Files. In Proceedings of the Asia-Pacific Advanced Network, KAIST Deajeon Korea, 19–23 August 2013; Volume 36, p. 17.
16. Khan, W.; Ullah, H.; Ahmad, A.; Sultan, K.; Alzahrani, A.J.; Khan, S.D.; Abdulaziz, S. CrashSafe: A formal model for proving crash-safety of Android applications. *Hum.-Cent. Comput. Inf. Sci.* **2018**, *8*, 21. [CrossRef]
17. Tiwari, P.; Tere, G.; Singh, P. Malware Detection in Android Application by Rigorous Analysis of Decompiled Source Code. In Proceedings of the International Conference on Computing Communication Control and Automation (ICCUBEA), Pune, India, 12–13 August 2016; pp. 1–6.
18. Bagheri, H.; Sadeghi, A.; Garcia, J.; Malek, S. Covert: Compositional analysis of Android inter-app permission leakage. *IEEE Trans. Softw. Eng.* **2015**, *41*, 866–886. [CrossRef]
19. Kitsaki, T.I.; Angelogianni, A.; Ntantogian, C.; Xenakis, C. A forensic investigation of Android mobile applications. In Proceedings of the 22nd Pan-Hellenic Conference on Informatics, Athens, Greece, 29 November–1 December 2018; pp. 58–63.
20. Tam, K.; Feizollah, A.; Anuar, N.B.; Salleh, R.; Cavallaro, L. The evolution of android malware and android analysis techniques. *ACM Comput. Surv.* **2017**, *49*, 76. [CrossRef]
21. TikTok and WhatsApp Retain Top Positions in App Store Download Ranking, BusinessofApps. Available online: https://www.businessofapps.com/news/tiktok-and-whatsapp-retain-top-positions-in-app-store-download-ranking/ (accessed on 28 October 2019).
22. 60+ Fascinating Smartphone Apps Usage Statistics For 2019 [Infographic]. Available online: https://www.socialmediatoday.com/news/60-fascinating-smartphone-apps-usage-statistics-for-2019-infographic/550990/ (accessed on 30 August 2019).