International Journal of ElectronicCommerce Studies
Vol.9, No.2, pp.209-238, 2018
doi: 10.7903/ijecs.1651

PERMISSION WATCHER TOOL: A SANDBOX TOOL-BASED STATIC AND DYNAMIC ANALYSIS FOR ANDROID APPS

Er-rajy Latifa
Cadi Ayyad University
Av Abdelkrim Khattabi, B.P. 511 - 40000, Morocco
Errajy.latifa@gmail.com

El Kiram My Ahmed
Cadi Ayyad University
Av Abdelkrim Khattabi, B.P. 511 - 40000, Morocco
kiram@uca.ma

ABSTRACT

Android security has become a very important issue with regard to mobile phone development: Android gives great freedom to developers to create and publish their apps for free in the PlayStore. The security mechanism of Android is based on an instrument that gives users the information about permissions that the application requests before installing it. This authorization system provides an overview of the application, and this can help to raise awareness of its risks. However, standard users still do not have enough information to understand clearly these requested authorizations and their implications on their security. In this article, we present a tool called "Permission watcher" that combines dynamic and static analysis. Our proposed tool allows users to install any application with only the necessary permissions instead of accepting all permissions requested or cancel the installation completely.

Keywords: Permissions; Applications; Security; Tool

1. INTRODUCTION

The Android market[1] has grown considerably since its inception in 2008. Application developers may be forced to develop applications for monetary gain by billing through ad revenue. In fact, developers can easily publish their applications in the Google PlayStore. On the other hand, the

online documentation for the Android API is incomplete, which makes the process of assigning an application more complicated[2].

To notify users of the privacy and security of their sensitive data when installing an application, Android uses mandatory access control (MAC) as a permissions system[3]. At the time of installation, an application must request authorization to access system resources (such as location, Internet, cellular network, etc.), and the user either accepts all the requested permissions or cancels the installation, since it is not possible to accept or deny access privileges selectively. Thus, many users simply accept these terms of use without taking into account their implications on their personal data. Such action can be very dangerous for their private data[4]. For example, if an application has granted certain critical permissions such as Internet permissions, this application can easily control communication with remote servers, and if it has access to the camera as well, it can send the user's personal pictures to any server on the Internet.

In May 2014, Google updated the Play Store to simplify the display of permissions for the user and help him or her to better understand their meanings. Google grouped the permissions by categories, and as a result, from over 150 permissions, we obtained a dozen categories, including one "other", which includes everything that does not fit elsewhere[5].

With the old system, in each update of the application, if the developer adds a new authorization, the Play Store displays it to the user who must accept it. With the new system, the developer can, for example, add the ACCESS_SUPERUSER permission that allows him or her to take control of all the phone's functions and storage, since it belongs to the category "other"[6].

In this paper, we present the Permission Watcher tool, which offers three mean stages: static analysis, dynamic analysis, and applications repackaging. After the user installs Permission Watcher on his or her Android device, the tool immediately inserts instrumentation code into arbitrary Android applications. The monitoring code then intercepts an application's interaction with the system in case of updates to enforce various security policies by watching these updates and the permissions that may be added without the user authorization.

The main advantage of the Permission Watcher tool is to enable the user to install an application with only the necessary permissions instead of accepting all the requested items or completely canceling the installation.

2. BACKGROUND

2.1 Android permission system

Android controls access to system resources by requiring permissions that the user must approve before authorizing the application installation using a bidirectional process. First, the developer defines the necessary permissions that are prerequisite to running the functionality of the application. Secondly, to start the installation, the user must approve without exception all the permissions required by the application[7]. The permissions requested by applications are divided into four levels:

(1) Normal - this level contains the permissions that protect access to API calls. These permissions are not dangerous, since they cannot cause actual harm to the user (for example, SET_WALLPAPER controls the possibility of modifying and changing the user's background) and of course, when the applications request them, they are automatically granted.

(2) Dangerous - this type of permissions controls access to API calls that are potentially dangerous, such as those related to the expense or collection of sensitive and/or private information; for example, dangerous permissions that have the purpose of sending text messages, reading the contact list, calling numbers, and opening Internet applications without the user's awareness.

(3) Signature - These permissions regulate access to the most dangerous privileges, such as the ability to control the backup process or remove application packages. They are automatically granted to an application when this application is developed by the same person, which means that it is signed with the same certificate. The purpose of this level is to allow applications that are linked or they are part of a suite to share data between them.

(4) Signature/System –This level shares the same features as Signature, except that the system image automatically obtains these permissions. This level of permissions was only created to be used by device manufacturers.

2.2 Application Installation Deconstruction

The Android Package Kit (APK) contains all the application code such as Dalvik EXecutable (DEX) file that runs on a Dalvik virtual machine, resources (not executable like graphics, multimedia files, user interface components, etc.), assets, certificates, and the manifest file. A user or developer can install an APK file directly on a device (i.e., not via a network upload) using a computer or communication program such as AdB or via an

application file manager in a process called sideloading. The APK file components are digitally marked with the developer's signature key.

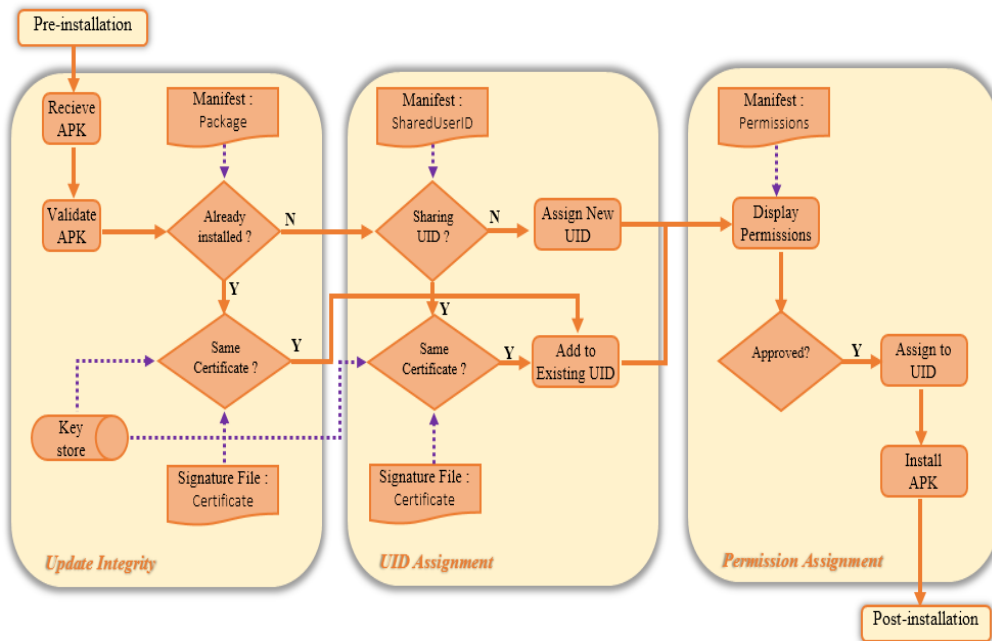

Figure 1. Abstract model of the Android installation process for an application package (apk)

The certificate by the developer may be self-signed and is contained within the package of application[8]. Any developer can create and distribute applications (even those who do not have a Google account) in the Google Play Store, through developer websites (side loading), or through third-party markets such as Appstore or Amazon. The absence of control over applications distributed through the Play Store shows the importance of enforcing the security measures within the Android operating system process. During the installation of a new application, permissions are approved prior to installation; however, the rest of the process remains the same. At the beginning, the application package validity is verified: the system ensures that the Android application package, since being signed, has not been corrupted or modified and that it comprises a valid certificate for the signing key. Android then decides whether the application should replace an existing application or become a new installation if the application being installed requires the same permissions and package attribute in the manifest file as another presently installed application. After that, Android will consider the installation like an update. Therefore, the

certificate (or group of certificates in the case of being signed by multiple keys) is matched with the certificate(s) of the application already installed. If both applications share the same key(s), at that point, the presently installed application is removed and the new application is installed in its place with preserving all user data from the removing one. Otherwise, the new application is installed as a primary installation. Afterward, Android should assign a UserID to the application (Figure 1). In this case, the UserID of previous application is used. If it is an initial installation, Android verifies whether the application manifest contains the directive sharedUserId. If it is so, Android searches for any other installed applications using the same key(s) for the signature that also have in their manifest a specified sharedUserId. If such applications are found, the application is assigned with the same UserID. Then, a new UserID is made. Lastly, permissions should be assigned to the UserID. The user is invited to approve the permission assignments after reviewing them before the application installation. In the case that a sharedUserId is not used, permissions recorded in the application manifest are assigned to the UserID. Once the sharedUserId is used, the UserID is assigned all permissions associated in the application manifests that share the UserID. If the application is updating an already installed application, the permissions listed in the updated application's manifest are assigned to the UserID[9].

3. PERMISSION WATCHER TOOL

3.1 Description

The Permission Watcher tool is a sandbox-based dynamic and static analysis that evaluates Android application permissions during installation time through several levels after using the APKtool[24], which is integrated in our proposed tool in order to extract the manifest file from the APK file. Figure 2 presents an example of the manifest extraction.

In addition, we used Java reflection[25] to get all API calls contained in the manifest file, which will be used in the static analysis phase. Generally, before installation, each application goes through two analysis levels: static and dynamic. The biggest advantage of the Permission Watcher tool is that it enables the user to install an application with only the necessary permissions instead of accepting all the requested permissions or completely canceling the installation.

```
$ apktool d test.apk
I: Using Apktool 2.2.4 on test.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: 1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
$ apktool b test
I: Using Apktool 2.2.4 on test
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Building apk file...
I: Copying unknown files/dir...
```

Figure 2.example of manifest extraction

3.2 How Permission Watcher work

The Permission Watcher tool has three major processes, as shown in Figure 3: static analysis, dynamic analysis, and applications repackaging. After the user installs Permission Watcher on his or her Android device, the tool immediately inserts the instrumentation code into arbitrary Android applications and the monitoring code that intercepts the application interactions with the system in case of updates in order to enforce the various security policies by watching these updates and the permissions that may be added without user authorization.

Static Analysis: The first thing that the tool does during the static analysis phase is to scan the Android application package (APK) for special patterns (for example, Runtime.Exec ()), which is used to classify the application to facilitate and speed up the reading of the database. Our implementation of static analysis is run offline, which makes it light enough to run on the Android device. However, dynamic analysis requires emulation on a more powerful machine.

When the user wants to install an application for the first time, the Permission Watcher tool preforms a static analysis that compares the permissions requested by the application with a database that contains that

lists the permission requirements for every API calls and permission specifications for more than one version of Android to make a clear decision about what kind of permissions are requested by the application (necessary, unnecessary, or dangerous) as illustrated in Figure 4. In fact, this database contains almost all API calls that can be required by Android applications.

We created one-to-many permission-API mappings manually by parsing the API documentation and inserting into the database several functions and permissions upon which the application depends in order to create a permission mapping which was quite complex in general.

For example, when using and instantiating a Bluetooth connection that requires BLUETOOTH permission, which is a fairly simple example, the LocationManager class cannot be instantiated directly, and the permission varies based on the constants used in the " instantiation.

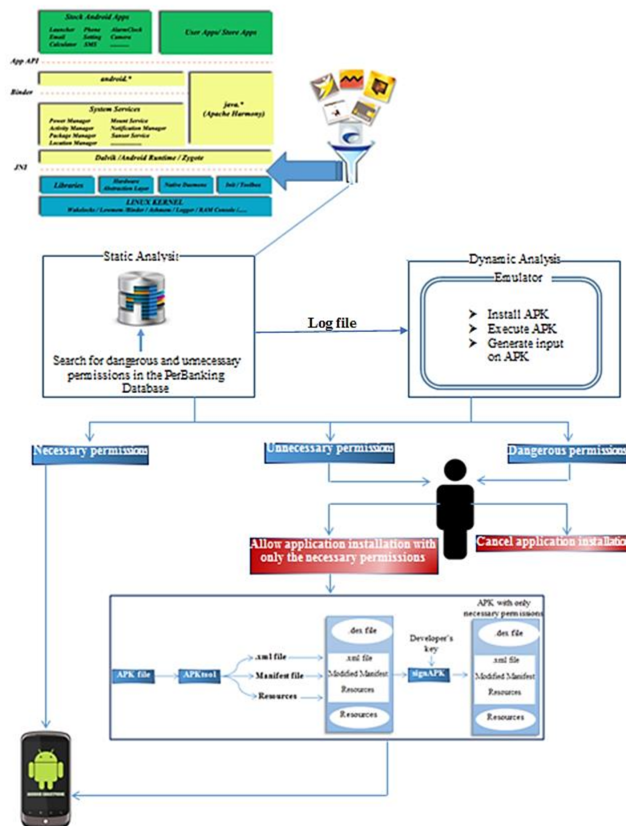

Figure 3. Permission Watcher Tool operating scheme

For example, if the application wants to use GPS_PROVIDER with LocationManager, it requests the ACCESS_FINE_LOCATION permission, and it must request permission for the ACCESS_COURSE_LOCATION in order to use the NETWORK_PROVIDER with LocationManager.

The Permission-API database is created as a priori and is simply loaded the first time the Permission Watcher tool is executed. Once created, the database should require little maintenance, since each database table is particular to an Android API revision, which is static. The only maintenance would be the result of an error or omission in the database itself. The existing documentation is inconsistent, which complicates the Permission-API database creation. Furthermore, in this static analysis, we used a method that examines the set of permissions declared by the application, especially those related to personal information such as credentials data, contacts, calendar events, email, and SMS/MMS. Then, the tool determines the permission requirement for every API by the application in order to produce a mapping of permissions that the application may need for its function.

rowid	Name	Group	Protection level	API l...	Description	Constant value
1	READ_CALENDAR	CALENDAR	Dangerous	1	Allows an application to read the users calendar d...	android.permission.READ_CALENDAR
2	READ_CALL_LOG	CONCAT	Dangerous	16	Allows an application to read the users call log.	android.permission.READ_CALL_LOG
3	READ_EXTERNAL_STORAGE	STORAGE	Dangerous	16	Allows an application to read from external storage.	android.permission.READ_EXTERNAL_STORAGE
4	READ_PHONE_NUMBERS	PHONE	Dangerous	26	Allows read access to the device s phone number(s)	android.permission.READ_PHONE_NUMBERS
5	READ_SMS	SMS	Dangerous	1	Allows an application to read SMS messages.	android.permission.READ_SMS
6	READ_VOICEMAIL	PHONE	signature|privileged	21	Allows an application to read voicemails in the system.	com.android.voicemail.permission.READ_VOICEMAIL
7	SEND_SMS	SMS	Dangerous	1	Allows an application to send SMS messages.	android.permission.SEND_SMS
8	GET_ACCOUNTS	ACCOUNTS	Dangerous	1	Allows access to the list of accounts in the Accounts S...	android.permission.GET_ACCOUNTS
9	INTERNET	NETWORK	NORMAL	1	Allows applications to open network sockets.	android.permission.INTERNET
10	ACCESS_COARSE_LOCATION	LOCATION	Dangerous	1	Allows an app to access approximate location.	android.permission.ACCESS_COARSE_LOCATION
11	ADD_VOICEMAIL	PHONE	Dangerous	14	Allows an application to add voicemails into the syste...	com.android.voicemail.permission.ADD_VOICEMAIL
12	ANSWER_PHONE_CALLS	PHONE	Dangerous	26	Allows the app to answer an incoming phone call.	android.permission.ANSWER_PHONE_CALLS
13	BIND_NFC_SERVICE	OTHERS	signature	19	Must be required by a HostApduService or OffHostA...	android.permission.BIND_NFC_SERVICE
14	CALL_PHONE	PHONE	Dangerous	1	Allows an application to initiate a phone call without ...	android.permission.CALL_PHONE
15	CAMERA	CAMERA	Dangerous	1	Required to be able to access the camera device.	android.permission.CAMERA
16	READ_CONTACTS	CONTACT	Dangerous	1	Allows an application to read the users contacts data.	android.permission.READ_CONTACTS
17	READ_PHONE_STATE	PHONE	Dangerous	1	Allows read only access to phone state, including the ...	android.permission.READ_PHONE_STATE
18	RECEIVE_SMS	SMS	Dangerous	1	Allows an application to receive SMS messages.	android.permission.RECEIVE_SMS
19	WRITE_CALENDAR	CALENDAR	Dangerous	1	Allows an application to write the users calendar data.	android.permission.WRITE_CALENDAR
20	WRITE_CALL_LOG	PHONE	Dangerous	16	Allows an application to write (but not read) the users...	android.permission.WRITE_CALL_LOG

Figure 4. Extract of database used in static analysis

In this analysis, we exploited Android functions and components such as Binder, Intents, Content Providers, and permission check functions that check for the presence of permission in order to increase the performance of this static analysis that consists of three phases:

(1) Decompression: As we mentioned before, an Android application is a compressed file (ZIP). When it is not compressed, its content is divided into three main parts:

- AndroidManifest.xml - An XML file that contains the application meta-information, such as descriptions, security permissions requested, etc.
- classes.dex – A single file that holds the complete bytecode interpreted by Dalvik VM.
- res / - A folder composed of files defining the layout, language, etc.

(2) Get the starter name: In this step, the tool extracts the main activity called "launchable activity" from the manifest file, which is not only needed to identify the application, but is also important later for dynamic analysis because it serves as an entry point for the user interface of the application.

(3) Decompilation: The classes.dex file is converted into human format using Baksmali.[26] This file holds the actual bytecode of the application. The decompilation produces a Java typical hierarchy of folders containing files with a pseudo-code that facilitates the analysis.

All permissions found are saved in a log file to be used in dynamic analysis.

Beside determining what permissions will be granted at the time of installation, additional information is extracted to better identify the malicious applications. This applies especially to the usage of:

- Native Java interface, which can be used to dynamically load native libraries.
- Sytem.getRuntime (). Exec (...), which can be used to generate processes of indigenous children and surpass the normal application life cycle.
- Reflection, which can be used to bypass API restrictions.
- Services and provision of the IPC, which can drain the battery or overload the CPU of the device

Dynamic analysis: This analysis is based on an Android virtual device based on QEMU[27], similar to the one provided with the Android SDK.

In this work, the application is installed in the standard Android emulator from the Google Android SDK. Once the application installation is complete, Monkey Tool[28],a program installed inside the emulator, generates a set of random pseudo-streams that present the user events such as clicks,

keys, gestures, and a number of system-level events. The Monkey was mainly invented for stress testing applications. Our tool is placed in the kernel space and shakes the system calls for logging. The dynamic analysis result helps us to record application behavior at the system level. The resulting log file will then be summarized and reduced to a mathematical vector for better analysis. The kernel module ensures that every occurrence of a system call is saved with the required permission. This ensures that the registration of a complete system state is fulfilled and that no malicious activity can be hidden. An application system call log is stored in a separate file.

Preparation and start of the emulator: As stated previously, we developed a mobile device emulator similar to the emulator Android SDK run on normal computers. This means that it supports Android virtual device (AVD) configurations used in applications testing. It removes all the hardware and software features of a typical mobile device, except phone calls. The running application inside the emulator can exploit the services of the Android platform to call other applications such as network access, provide audio and video playback, store and retrieve data, inform the user, make transitions and graphic themes, simulating latency effects, and packets on the data channel, receiving SMS messages or phone calls.

The purpose of dynamic analysis is to examine the system state changes that occur when a given application is run. To fulfill this aim, the emulator has a policy called log-only that does not actively intercept the system state change. We have developed a loadable kernel module (LKM)[29] that applies the policy set by the sandbox environment. Insertion of the LKM into the running kernel of the Android device emulator is done with the Android Debugging Bridge (ADB)[30]that accompanies the Android SDK. Once the LKM is loaded, the generated output is sent to a log file.

Install APK and start Monkey: We used ADB to install the APK of the application inside the emulator. The ADB copies the APK file into the emulator, and then it runs PackageManager, which presents as essential part of Android. Finally, ADB installs the application inside the virtual emulator. This means that the APK will be decompressed and copied into the specified directories. After installation, the application is started automatically.

Get system call logs: Inside the emulator, the monkey simulates human interaction with the application being examined.

During the execution time for the monkey, there are exactly 500 events generated with a silence of 1000ms between each two events. When the monkey is finished, the mobile device emulator process is killed and the

used AVD setup that was created for the running application is deleted.

This emulator uses an API tracer to monitor how Java application components communicate with the Android Java framework, how its native components interacts with the system, and how its Java components and native components communicate through the interface JNI. The native instruction tracer and the Dalvik instructions tracer embedded in the emulator's code source examines how a malicious application (based on the necessary and dangerous permissions obtained after the comparison between the log file and the one obtained from the Static analysis) behaves internally by recording detailed instructions.

The Dalvik Tracer stores bytecode statements for malicious Java components, and the native instruction tracer stores machine-level instructions for native components (if any). Taint tracking observes how the malicious software obtains and discloses sensitive information (eg, GPS Location, IMEI, and IMSI) using the spoofing analysis component in the emulator. This virtual environment uses a method called "API hooking" in order to monitor the behavior of the analyzed application by intercepting function calls. These hooks are generally inserted during runtime. However, they can also begin working before the application execution. The physical change can accomplish hooking with binary rewriting or changing the API to monitor function calls before execution. This way, the emulator can easily detect if the application has any bad effects on user data.

The following scheme abstracts in brief how the emulator works (Figure 5):

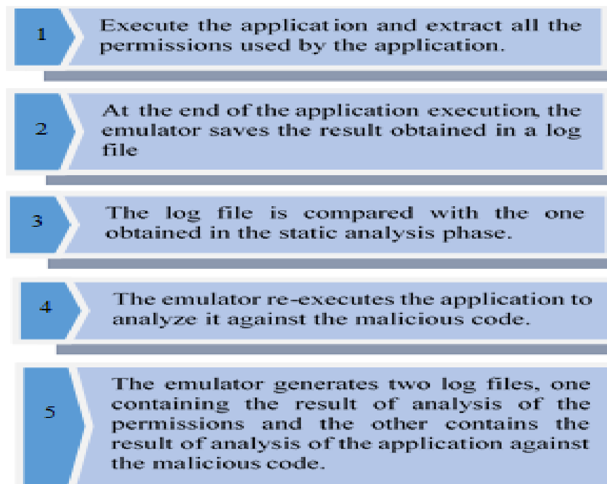

Figure 5. virtual emulator work

At the end of the dynamic analysis, the Permission Watcher Tool receives two logs files. It examines first the result of the code analysis. If the application contains malicious code, its installation is automatically aborted, and a popup window is shown to the user explaining why the application installation has been cancelled. Otherwise, the Permission Watcher Tool examines the permission result if there are unnecessary or dangerous permissions requested by the application, and then the tool sends the user a notification that invites him either to allow application installation with only the necessary permission or to cancel the installation completely. In the case in which the user chooses to continue the installation, the tool repackages the application in order to delete the unnecessary permissions before allowing its installation.

Application repackaging: In this process (Figure 6), we have to delete the unnecessary and dangerous permissions from the application manifest file.

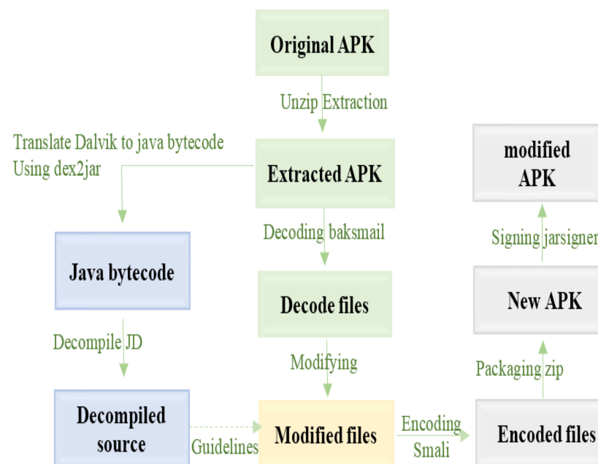

Figure 6. Application repackaging steps

Each application goes through five steps before it is ready to be installed on the user's device, because the code in the APK file is so difficult to read by a human, since it contains Dalvik bytecode (Dex format): extraction, decoding, modifying, encoding, and packing. Figure 7 shows a rough overview of the process used to modify an existing .apk file. The purpose of the extraction and decoding steps is to transform an .apk file into an easily editable form. The modification step is an application-specific step which involves reading and modifying the bytecode. During the encoding and packing steps, a new .apk file is created from the modified files.

Extraction: The extraction step involves separating an .apk file into multiple files. Since .apk files are based on the JAR specification, they can be extracted with any zip-based compression utility such as winZip[31]. As part of the extraction process, the META-INF directory is deleted. This directory contains various files used to verify the JAR integrity. Since the manifest file, which is among the archive contents, will be edited, the files inside META-INF directory will need to be recreated.

Decoding: During this step, human readable versions of the binary files are created. So, files with a .dex extension containing Dalvik bytecode are converted into an equivalent text format using Baksmali disassembler. This tool disassembles a .dex file into multiple .smali files. Each file is a single java class. Android uses a binary XML format to speed up the application loading process. Therefore, before any reading or modifying is completed, these XML files must be converted into an equivalent text representation using the AXMLPrinter2 utility[32].

Modifying: In the modification step, we only modify the manifest file by deleting the unnecessary permissions described in the log file we got from the dynamic analysis step. For the Java code source generation, a standard Java decompiler called dex2jar tool is used.

Encoding: This step is similar to the decoding step. First, all modified manifest.xml file must be covered back into its binary formats. Then, a new classes.dex needs to be created from the modified .smali files. This step is performed using the smali assembler, which assembles a directory with all .smali files into a single .dex file.

Packaging: This step is based on the standard Android build process[33]. Firstly, application files, such as assembled .dex files, .xml binaries, and application elements, are stored in a zip archive. Android requires that all applications be cryptographically signed with an RSA certificate. Android uses these signatures to verify the integrity and the author of the archive. The Android installation process will reject all unsigned .apk files. The process of signing an .apk file is based on the JAR signature process[34]. Then, the jarsigner[35] utility is used to sign the modified .apk file with RSA certificates. Self-signed certificates are valid only during development. The final packaging step aligns the contents of the .apk file to a 32-bit limit. Zip alignment is performed with the zipalign[36]utility.

```
#create temporary folder
mkdir build
#copy everything
cp -r * build
rmdir build/build
#rebuild modified classes.dex
rm build/classes.dex
java -jar smali.jar -o build/classes.dex out
#rebuild modified AndroidManifest.xml
rm build/AndroidManifest.xml
AXmlEncode DecodedManifest.xml AndroidManifest.xml
#create apk
cd build
zip -r ../temp.apk *
cd ..
#(optional) keytool -genkey -keyalg rsa -alias MyCert
jarsigner temp.apk MyCert
#aling apk on 32-bit a boundary
zipalign 4 temp.apk build.apk
```

Figure 7. Typical coding and packaging script

This restructuring allows Android to directly memorize the archive sections to improve their performance. Although this step is not required, official documentation recommends aligning zip to all .apk files. Figure 6 shows a typical coding and packaging script. The script copies all the application's resources to a new temporary build directory. A close utility is used to create a new .apk file from the temporary directory. The archive is then signed with a private RSA certificate. Finally, the signed .apk is aligned with a 4-byte boundary with the zipalign utility.

4. EXPERIMENTS

To proof the correct working of the whole system, we analyzed an Android malware family, assuming that the permissions requested by these applications can be used to detect malware families.

In this test, we have specifically targeted the DroidDream[37] family as a test case to see if we can identify malicious software in this family as malicious depending on the requested permissions. DroidDream appeared for the first time in 2011 in the Google Play store, and there are several versions of this malware, which gave us a "family" of malware that has evolved and expanded in functionality from the basic version. Although all iterations have a similar name, they are completely different in their malicious techniques and objectives. Among DroidDream versions, there is a family called DroidDreamLight[38] that does not need user intervention to run. This malware successfully obtains root privileges on the user device and uses it to collect and send the user's personal information to a remote

server. Then it makes the victim download and install new malicious applications.

We worked with the teen DroidDream family obtained from the Android Malware Genome Project[39]. Table I presents the selected family.

4.1 Static analysis results

Permissions: As we explained before, the goal of static analysis is to identify permissions by examining the AndroidManifest.xml file. It begins by examining the permissions in the DroidDream dataset to determine the frequency of each occurrence. This analysis concentrates especially on the permissions that appeared in a super majority of DroidDream's APK files. By looking at the log file, we found that the following permissions were retrieved:

- CHANGE_WIFI_STATE - This permission allows the application to change the state of WiFi connectivity.
- ACCESS_WIFI_STATE - This permission allows the application to access information about the WiFi network.
- INTERNET - this permission permits the application to access and open the network sockets.
- READ_PHONE_STATE - This permission permits the application to access the phone state, but it isread-only.
- READ_CONTACTS and WRITE_CONTACTS - These two permissions allow the application to read and write the contact list found in the mobile phone.
- READ_LOGS - This permission permits the application to read log files of low-level system.
- ACCESS_NETWORK_STATE - This permission permits the application to access the network information.

The permissions found were saved in a log file and sent to the virtual emulator for the comparison with the one obtained after preforming the dynamic analysis. This step is more complicated than the static analysis and gives more detail about the nature of the analyzed application.

Table 1. Malwares family analyzed

Family	Description
DroidDrem	Botnet, it gained root access
FakeInstaller	Server-side polymorphic family
Plankton	It use class loading to forward details
DroidKungFu	It installs a backdoor
GinMaster	Malicious service to root devices
BaseBridge	It sends information to remote server
Adrd	It sends info to premium-rate numbers
Kmin	It sends info to premium-rate numbers
Geinimi	First Android botnet
Opfake	First Android polymorphic malware

4.2 Dynamic analysis results

By executing our set of DroidDream applications in the virtual emulator, we obtained some very interesting information about the entire code, including the usage permissions requested. That information helps us to understand malware behavior.

Services: A service is simply an application component that is capable of running long-term applications in the background without providing an interface to the user, and it is also capable of continuing to run in the background, even though the user closes the application and switches to another application.

In the analysis result, we found that 15 instances of DroidDream uses two services known as malware added to the Android. These services are services.com.android.root.AlarmReceiver and com.root.Setting. Figure 8 provides more information below.com.root.Setting decrypts a byte buffer using an XOR with a predefined key in the adbRoot class. The server IP address and its URL link are already decrypted in the byte buffer. This server is used for the data publishing on the infected phone on which the malware is installed in XML format using an HTTP POST request.

```
<service android:name="com.android.root.Setting" android:process=":remote" />
<service android:name="com.android.root.AlarmReceiver" android:process=":remote2" />
```

Figure 8. Malware services found

In addition to the malicious services added to trojanized packages, there is also a set of files added to the package assets. Assets include 3

native ARM applications, two are privilege escalation exploits and an application that allows it to execute shell commands as root.

If the exploit was successful, the Trojan attempts to install an additional package included in the malware assets in the form of sqlite.db. It contains a code that allows it to send extra information about the device victim and to download additional content.

Function names: We have programmed the virtual emulator in such a way that it automatically excludes the onlick, onCreate, and onDestroy functions, as they are generally exploited by Android applications, which does not show any potential malicious activity. The emulator also does not consider functions with obscene names: a, b, c, d, e, etc. which leaves the emulator eleven functions by which to check and detect their existence in the supermajority of code (Figure 9 illustrates an example of the functions detected by dynamic analysis). For example, the emulator can detect the following functions:

- getIMEI, getIMSI and getRawResource allow the application to collect user information.
- Installsu gains root permission on the victim device.
- sPackageInstalled examines whether an additional packet is installed or not.
- onReceive collects additional information on the network.
- PostUrl is dangerous because it can publish a URL.
- changeWiFiState and restoreWiFiStateconnect to WiFi without the knowledge of the user.
- removeExploit raises a red flag for its ability to exploit user information.

```
private static final int PERMISSIONS_REQUEST_READ_PHONE_STATE = 999;

private TelephonyManager mTelephonyManager;

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    if (checkSelfPermission(Manifest.permission.READ_PHONE_STATE)
            != PackageManager.PERMISSION_GRANTED) {
        requestPermissions(new String[]{Manifest.permission.READ_PHONE_STATE},
                PERMISSIONS_REQUEST_READ_PHONE_STATE);
    } else {
        getDeviceImei();
    }
....

@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
                                       int[] grantResults) {
    if (requestCode == PERMISSIONS_REQUEST_READ_PHONE_STATE
            && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        getDeviceImei();
    }
}

private void getDeviceImei() {

    mTelephonyManager = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
    String deviceid = mTelephonyManager.getDeviceId();
    Log.d("msg", "DeviceImei " + deviceid);
}
```

Figure 9. Example of functions detected by dynamic analysis

In the DroidDream samples that the emulator analyzed, the scan result shows that malware cannot be started automatically, so it requires that the user manually launch the infected application. Once the user runs the application, the DroidDream family transfers to a remote command and control server to access the user's sensitive information that includes the following:

- IMEI
- IMSI
- Device model
- SDK Version

The DroidDream family configuration allows it to perform at least one successful check with the command and control server that will respond and recognize the presence of malicious software on the infected user's device. Examination of the code by the emulator shows that the authors of the DroidDream family configured the malicious software in order to ensure

that the device is not yet infected by another variant of DroidDream. Then, the malware will not infect the device again if it is already infected. Analysis of the code revealed a very dangerous detail; DroidDream contains malicious code that allowed it to do most of its malicious work between 11 pm and 8 am, because most people sleep during that interval of time, and phones are less often used which makes it very difficult to detect that abnormal applications are running on the infected device.

Comparison of two log files: The result of the two log files comparison shows that almost all teen malware families do not need any of the requested permissions; they are only used to perform malicious activities. Our set of DroidDream versions collects users' IMEI and IMSI and sends them through the URL network socket connection.

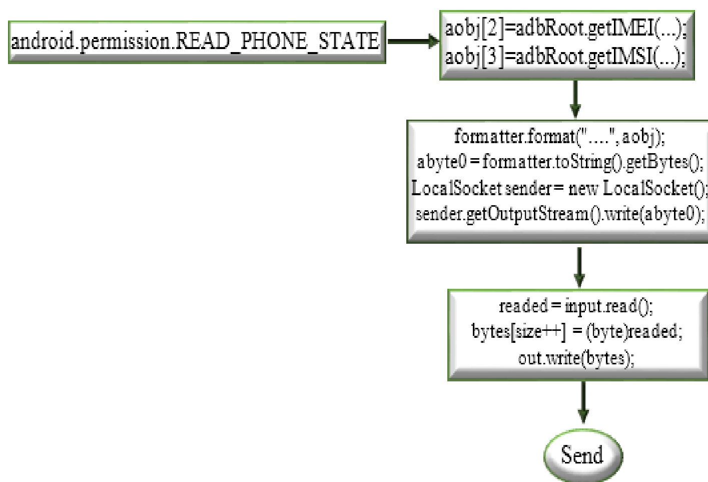

Figure 10. How DroidDream collects users' IMEI and IMSI, and sends them through URL network socket connection

This is a behavior of personal information stealing. URL network socket connection needs INTERNET permission, which is too conspicuous and may be easily caught by traditional methods. Figure 10 shows how DroidDream malware family obtains IMEI and IMSI and sends them through a Local Socket. So, the malware programs without READ PHONE STATE permissions receive the data from a same SOCKET ADDRESS of Local Socket and send them to a distant server. The user-sensitive data can also be transmitted through other public interfaces.

The installation of our set is automatically cancelled because it contains malicious code. However, we decided to test manually the repackaging

process of DroidDream.

4.3 Repackaging the DroidDream application

As previously stated, the goal of the repackaging process is to create a safe application with only the necessary permissions. To achieve this, we extracted, disassembled, patched, and reassembled the application.

Extracting and decoding: The extracting and decoding steps on DroidDream-infested applications is slightly different than on regular applications. The DroidDream application contains a nested .apk file in the assets/directory (under the name sqlite.db). Therefore, the extracting and decoding steps must also be performed on the nested .apk file.

Modifying manifest file: In the above manifest file, we deleted three permissions: READ_PHONE_STATE, READ_CONTACTS, and WRITE_CONTACTS

Rebuild APK: We use APKTool again to generate a new APK file.

Sign the APK file: Android requires all apps to be digitally signed before they can be installed. This requires each APK to have a digital signature and a public key certificate. The certificate and the signature help Android to identify the author of an app. From a security perspective, the certificate needs to be signed by a certificate authority, who, before signing, needs to verify that identify stored in the certificate is indeed authentic. Getting a certificate from an accepted certificate authority is usually not free, so Android allows developers to sign their certificates using their own private key, i.e., the certificate is self-signed. The purpose of self-signed certificates is that it allows apps to be run on Android devices, not for security. Developers can put any name they want in the certificate, regardless of whether the name is legally owned by others or not, because no certificate authority is involved to check it. Obviously, this entirely defeats the purpose of certificate and signature. Google Play Store performs some name verification before accepting an app, but other third-party app markets do not always conduct such verification. The entire process consists of three steps:

- Step 1: Generate a public and private key pair using the keytool[40].
- Step 2: Use jarsigner to sign the APK file using the key generated in the previous step.
- Step 3: Install the modified application on the user device.

4.4 Statistic

We applied our tool to 100 mobile banking applications from the Google Play Store. 50 of these applications are Moroccan banks, 30 are Tunisian, and 20 are Algerian.

Unnecessary permission: Permissions Watcher identified that 45% of applications have unnecessary permissions, which can be dangerous. Table 2 below shows that almost all unnecessary requests by applications in our set are dangerous.

Dangerous permissions: We focused on the prevalence of dangerous permissions. As we mentioned before, dangerous permissions are displayed as a warning to users during the applications installation and may have serious security ramifications on the user's personal data. We noted that 82% of the applications analyzed have at least one dangerous permission. Permissions in Android are grouped into feature categories. This provides a relative measure of part of the protected API that is used by the applications.

A small number of permissions are required very frequently.

In particular, the INTERNET permission is strongly used to bind the user to his bank to benefit from the mobile services available to him. We find that 24% of applications request INTERNET as their only dangerous permission. We also found that 38% of applications combine between three dangerous permissions. For example, 16% of applications require the following permissions: CONTACTS, SMS, and PHONE at the same time, which means that these applications have the power to control and use the mobile phone to call numbers and send SMS messages without the user's awareness, which could lead to dangerous consequences during the user’s banking transactions.

Table 2. The most common unnecessary permissions requested

PERMISSION	%	PERMISSION LEVEL
ACCESS_NETWORK_STATE	25%	Normal
READ_PHONE_STATE	45%	Dangerous
ACCESS_WIFI_STATE	38%	Normal
WAKE_LOCK	5%	Dangerous
WRITE_EXTERNAL_STORAGE	27%	Dangerous
ACCESS_LOCATION	36%	Dangerous
PHONE	55%	Dangerous
SMS	65%	Dangerous
CAMERA	15%	Dangerous
INTERNET	85%	Dangerous
CONTACTS	30%	Dangerous
DEVICE ID & CALL INFORMATION	28%	Dangerous
PHOTOS/MEDIA/FILES	34%	Dangerous

Although many applications ask for at least one dangerous permission, the total number of permission requests is typically low. The most highly privileged application in our set asks for less than half of the available 56 dangerous permissions. Figure 11 shows the distribution of dangerous permission requested.

Several important categories are requested relatively infrequently, which is a positive finding. Permissions in the PERSONAL_INFO and COST_MONEY categories are only requested by 5% of applications. The PERSONAL_INFO category includes permissions associated with the user's contacts, calendar, etc. COST_MONEY permissions let applications send text messages or make phone calls without user confirmation. Users have reason to be suspicious of applications that ask for permissions in these categories.

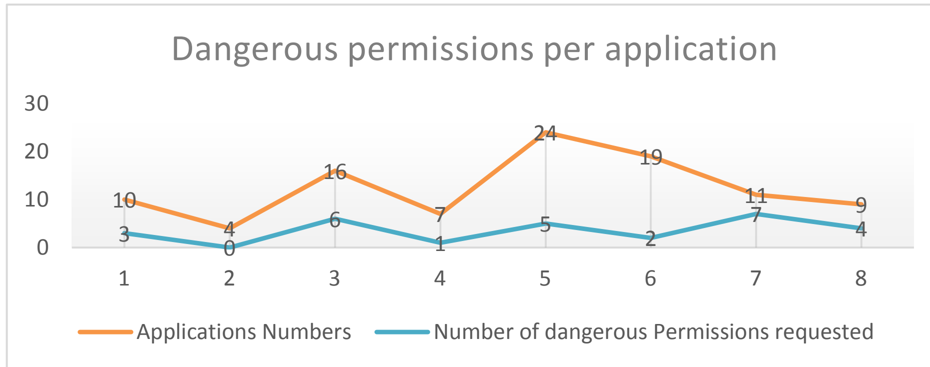

Figure 11. Dangerous permissions per application

Table 3 shows the percentage of dangerous permissions requested in each category. Nearly all applications (82%) ask for at least one dangerous permission, which indicates that users frequently install applications with dangerous permissions.

We were interested in the dangerous permissions most frequently requested by all the banking applications we analyzed. Figure 12 below illustrates the results of the analysis obtained.We notice the following permissions: INTERNET(NETWORK),WRITE_EXTERNAL_STORAGE (STORAGE), READ_PHONE_STATEMENT, and WAKE_LOCK are the most frequent dangerous permissions requested.

WAKE_LOCK permission allows an application the useof PowerManager WakeLocks to keep processor from sleeping or screen from dimming. Thismeans that suchpermission is totally unnecessary for a mobile banking application. The same is truefor the permission ACCESS_FINE_LOCATION that allows an application to access the precise location of the mobile device owner.

Table 3. Applications with at least one dangerous permission in each category

CATEGORY	APPLICATIONS %
NETWORK	66 %
SYSTEM_TOOLS	39.7 %
STORAGE	34.1 %
LOCATION	26%
PHONE_CALLS	35%
PERSONAL_INFO	13%
HARDWARE_CONTROLS	17%
COST_MONEY	9%
MESSAGES	5%
ACCOUNTS	2%
DEVELOPMENT_TOOLS	0%

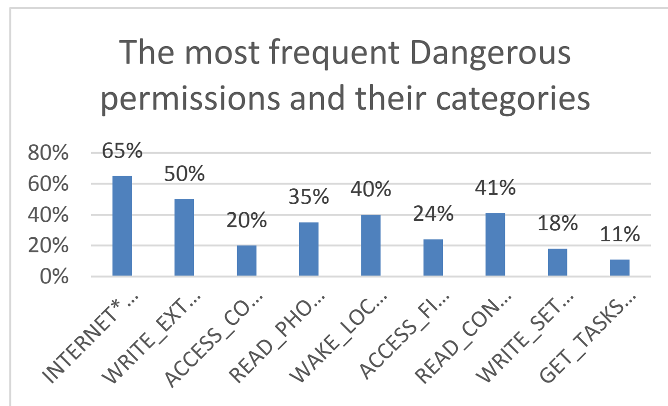

Figure 12. Dangerous permissions per application

5. CONCLUSION

In this paper, we present a tool called Permission Watcher that analyzes the permissions requested by Android applications at the time of installation and after their updates. Our reference implementation is very efficient and induces a small performance overhead. Therefore, we have developed this tool especially for users without a technical and security background. Our aim was to create system-based permissions on a stable footing by

informing users about dubious permission sets and gives them a third option when installing applications.

6. REFERENCES

[1] M. Amir, *Energy-aware location provider for the Android platform*. University of Alexandria, 2010.
[2] Scientia mobile, *Mobile overview report October - December 2014*. Retrieved on June 5, 2016, from https://www.scientiamobile.com/movr-mobile-overview-report/.
[3] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, Android Permissions: User attention , comprehension , and behavior. *Proceedings of the Eight Symposium on Usable Privacy and Security*. 2012. http://dx.doi.org/10.1145/2335356.2335360.
[4] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, L4Android: a generic operating system framework for secure smartphones. *Proceedings of the 1st ACM workshop on Security and privacy in smartphones mobile devices* (p39–50). 2011. http://dx.doi.org/10.1145/2046614.2046623.
[5] N. Viennot, E. Garcia, and J. Nieh, A measurement study of google play. *Proceedings of the 2014 ACM International conference on Measurement and modeling of computer systems* (p221–233). 2014. http://dx.doi.org/10.1145/2637364.2592003.
[6] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, Upgrading your Android, elevating my malware: privilege escalation through mobile OS updating. *Proceedings of 2014 IEEE Symposium on Security and Privacy*. 2014. http://dx.doi.org/10.1109/SP.2014.32.
[7] Z. Fang, W. Han, and Y. Li, Permission based Android security: Issues and countermeasures. *Computer Security*, 43, p205–218, 2014. http://dx.doi.org/10.1016/j.cose.2014.02.007.
[8] I. R. Forman, and N. Forman, *Java Reflection in Action*. Manning Publications, 2004.
[9] J. Butler, VICE – Catch the hookers! *Black Hat USA 61*, p17-35, 2004.
[10] P. Reviewed, and K. Anne, *Connecting Math Methods and Student Teaching through Practice-Based Strategies: A Study of Pre-Service Teachers' Math Instruction.* Electronic Thesis and Dissertations UCLA, 2012.
[11] R. Di Pietro, F. Lombardi, and S. Rossicone, *Modeling Mobile Resource Security*. Mat.Uniroma3.It, 2013.
[12] T. Report, *Analysis of Dalvik virtual machine and class path library.* Retrieved on June 7, 2016, from http://lim.univ-reunion.fr/staff/fred/Doc/Dalvik/Analysis-of-Dalvik-VM.pdf.

[13] D. Barrera, *Securing decentralized software installation and updates*. Carleton University, 2014. https://doi.org/10.22215/etd/2014-10421.

[14] W. Enck, D. Octeau, P. Mcdaniel, and S. Chaudhuri, A study of Android application security. *Proceedings of the 20th USENIX Conference on Security* (p21).2011.

[15] A. P. Felt, K. Greenwood, and D. Wagner, The effectiveness of application permissions. *Proceedings of the 2nd USENIX Conference on Web Application Development* (p7), 2011.

[16] W. Enck, M. Ongtang, and P. McDaniel, On lightweight mobile phone application certification. *Proceedings of the 16th ACM conference on Computer and Communications Security* (p235-245). 2009. http://dx.doi.org/10.1145/1653662.1653691.

[17] D. Barrera, H. G. Kayacik, P. C. Van Oorschot, and A. Somayaji, A methodology for empirical analysis of permission-based security models and its application to Android. *Proceedings of the 17th ACM Conference on Computers and Communications Security* (p73–84). 2010. http://dx.doi.org/10.1145/1866307.1866317.

[18] A. Möller, T. U. München, F. Michahelles, S. Diewald, L. Roalter, and M. Kranz, Update behavior in app markets and security implications: A case study in google play. *Proceedings of the 3rd International Workshop on Research in the Large*, *Held in Conjunction with Mobile HCI* (p3–6), 2012.

[19] L. Tenenboim-Chekina, O. Barad, A. Shabtai, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, Detecting application update attack on mobile devices through network features. *Paper Presented at 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Turin, Italy, April 14-19, 2013. http://dx.doi.org/10.1109/ INFCOMW.2013.6970755.

[20] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, Measuring user confidence in smartphone security and privacy. *Proceedings of the Eighth Symposium on Usable Privacy and Security*. 2012. http://dx.doi.org/10.1145/2335356.2335358.

[21] B. Liu, J. Lin, and N. Sadeh, Reconciling mobile app privacy and usability on smartphones: could user privacy profiles help? *Proceedings of the 23rd international conference on World wide web* (p 201–212). 2013. http://dx.doi.org/10.1145/2566486.2568035.

[22] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, A conundrum of permissions: installing applications on an Android smartphone. *International Conference on Financial Cryptography and Data Security* (p68-79). 2012.

[23] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. *Technische Universität Darmstadt, Technical*

Report TR-2011-04(p4–7). 2011.

[24] R. Xu, H. Saïdi, R. Anderson, and H. Saıdi, Aurasium: practical policy enforcement for Android applications. *Proceedings of the 21st USENIX Conference on Security Symposium* (p27). 2012.

[25] M. Zhang and H. Yin, AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. *Proceedings 2014 Network and Distributed System Security Symposium* (p23–26). 2014. http://dx.doi.org/10.14722/ndss.2014.23255.

[26] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, Detecting repackaged smartphone applications in third-party android marketplaces. *Proceedings of the second ACM Conference on Data and Application Security and Privacy* (p317–326). 2012. http://dx.doi.org/10.1145/2133601.2133640.

[27] J. Kornblum, Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, p 91–97, 2006.

[28] I. You and K. Yim, Malware obfuscation techniques: A brief survey. *Proceedings of the 2010 International Conference on Broadband, Wireless Computer, and Communication and Applications* (p297–300). 2010. http://dx.doi.org/10.1109/BWCCA.2010.85.

[29] A. Desnos, and G. Gueguen, Android: From reversing to decompilation. *Proceedings of the Black Hat Abu Dhabi* (p1–24). 2011.

[30] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, SCanDroid: Automated security certification of Android applications. *University of Maryland Department of Computer Science, Technical Report CS-TR-4991*(p238).2010.

[31] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2), p393-407, 2014. http://dx.doi.org/10.1145/2619091.

[32] V. Rastogi, Y. Chen, and W. Enck, AppsPlayground: Automatic security analysis of smartphone applications. *Proceedings of the third ACM Conference on Data and Application Security and Privacy* (p209–220). 2013. http://dx.doi.org/10.1145/2435349.2435379.

[33] N. J. Percoco, and S. Schulte, Adventures in BouncerLand. Retrieved on June 8, 2016, from https://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf.

[34] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, Vetting undesirable behaviors in android apps with permission use analysis. *Proceedings of the 2013 ACM SIGSAC Conference on*

Computer & Communications Security (p611–622). 2013. http://dx.doi.org/10.1145.2508859.2516689.

[35] J. Andrus, C. Dall, A. Van Hof, O. Laadan, and J. Nieh, Cells: A virtual mobile smartphone architecture categories and subject descriptors. *Proceedings of the Twenty-Third ACM Symposium on Operating System Principles* (p173–187). 2011.

[36] P. Zhang, H. Sun, and Z. Yan, Mechanism for security enhancement in mobile application installation. *Proceedings of the 2012 2nd International Conference on Computer and Information Applications* (p 4382–4387). 2013. http://dx.doi.org/10.2991/iccia.2012.81.

[37] K. Tam, S. J. Khan, A. Fattoriy, and L. Cavallaro, CopperDroid: automatic reconstruction of Android malware behaviors. *Proceeding 2015 Network and Distributed System Security Symposium*.2013. http://dx.doi.org/10.14722/ndss.2015.23145.

[38] T. Vidas, and N. Christin, Evading Android runtime analysis via sandbox detection. *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (p447–458). 2014. http://dx.doi.org/10.1145/2590296.2590325.

[39] S. Neuner, V. Van Der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl, Enter Sandbox: Android Sandbox Comparison. *Proceedings of the Third Workshop on Mobile Security Technologies (MoST)*. 2014.

[40] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy Magazine*, 5(2), p32–39, 2007. http://dx.doi.org/10.1109/MSP.2007.45.

[41] A. Dewald, T. Holz, and F. C. Freiling, ADSandbox. *Proceedings of the 2010 ACM Symposium on Applied Computing* (p1859-1864). 2010. http://dx.doi.org/10.1145/1774088.1774482.

[42] X. Zhang, F. Breitinger, and I. Baggili. Rapid Android parser for investigating DEX files (RAPID). *Digital Investigation*, 17, p28–39, 2016. http://dx.doi.org/10.1016/j.diin.2016.03.002.

[43] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. Amorim, and M. D. Ernst, Static analysis of implicit control flow: Resolving Java reflection and Android intents. *2015 30th IEEE/ACM International Conference on Automated Software Engineering* (p669-679). 2015. http://dx.doi.org/10.1109/ASE.2015.69.

[44] J. Xu, S. Li, and T. Zhang, Security analysis and protection based on Smali injection for Android applications. *International Conference on Algorithms and Architectures for Parallel Processing* (p577–586). 2014. http://dx.doi.org/10.1007/978-3-319-11197-1_44.

[45] J. Ding, P. Chang, W. Hsu, and Y. Chung, PQEMU: A parallel system emulator based on QEMU. *2011 IEEE 17th International Conference on Parallel and Distributed Systems* (p176-283). 2011.

http://dx.doi.org/10.1109/ICPADS.2011.102.
[46] A. Developers, *UI/Application Exerciser Monkey*. Retrieved on June 8, 2016, from http://developer.android.com/%0Atools/help/monkey.html.
[47] K. Jones, Loadable kernel modules. *Usenix Magazine*, 26(7), p43-49, 2001.
[48] Google, *Android debugger bridge*. Retrieved on June 9, 2016, from https://developer.android.com/studio/command-line/adb.
[49] T. Kohno, Attacking and repairing the WinZip encryption scheme. *Proceeding of the 11th ACM Conference on Computer and Communications Security* (p72-81). 2004. http://dx.doi.org/10.1145/1030083.1030095.
[50] Android4me, *AXMLPrinter2.jar*[Online]. Retrieved on June 10, 2016, from https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/android4me/AXMLPrinter2.jar.
[51] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, Cuckoo: A computation offloading framework for smartphones. *International Conference on Mobile Computer, Application, and Services* (p59-79). 2012. https://doi.org/10.1007/978-3-642-29336-8_4.
[52] M. Zheng, M. Sun, and J. C. S. Lui, DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate Android malware. *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (p163-171). 2013. http://dx.doi.org/10.1109/TrustCom.2013.25.
[53] V. Jaglan, S. Dalal, and S. Srinivasan. Enhancing security of agent-oriented techniques programs code using jar files. *International Journal on Computer Science and Engineering*, 3(4), p1627–1632, 2011.
[54] S. R. Kurhade, and N. D. Gite, Android anti-malware analysis. *International Journal of Advanced Research in Computer Engineering & Technology*, 4(5), p2261–2266, 2015.
[55] F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone, Twinkle Twinkle Little DroidDream, How I Wonder What You Are? *2017 IEEE International Workshop on Metrology for Metrology for AeroSpace* (p 21-25). 2017. http://dx.doi.org/10.1109/MetroAeroSpace.2017.7999579.
[56] M. Balanza, K. Alintanahin, O. Abendan, and J. Dizon, DroidDream light lurks behind legitimate Android Apps. *2012 6th International Conference on Malicious and Unwanted Software* (p73-78). 2011. http://dx.doi.org/ 10.1109/MALWARE.2011.6112329.
[57] Y. Zhou, and X. Jiang, Dissecting Android malware: Characterization and evolution. *2012 IEEE Symposium on Security and Privacy* (p 95-109). 2012. http://dx.doi.org/10.1109/SP.2012.16.
[58] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, W. Menzel, W.

Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, The KeY tool Integrating object oriented design and formal verification. *Software System Model*, 4(1), p32-54, 2005. https://doi.org/10.1007/s10270-004-0058-x.