Android Apps Risk Evaluation: a methodology

EAI Endorsed Transactions
on Ubiquitous Environments

Research Article ICST.ORG

Android Apps Risk Evaluation: a methodology

Andrea Atzeni[1], Tao Su[1,*], Madalina Baltatu[2], Rosalia D'Alessandro[2], Giovanni Pessiva[3]

[1]DAUIN, Politecnico di Torino - Corso Duca degli Abruzzi, 24 - 10129 Torino, ITALY
[2]Telecom Italia Information Technology Security Lab, Via Reiss Romoli, 274, Torino, ITALY
[3]Reply SpA, Via Cardinale Massaia, 83, Torino, ITALY

Abstract

Android uses a permission-based security model to limit its app's capability. However, the user's decision is almost completely unrelated to the app's risk level due to insufficient information. The platform openness and the plethora of available software also make dangerous apps (not necessarily malware) very common.

To enhance end-user security awareness, we propose a new approach and tool to evaluate the potential risks of Android app packages. We integrated various static and dynamic analysis techniques into a framework able to detect suspicious activities, map them to fine-grained risk categories and evaluate them with the fuzzy logic algorithm. This tool can retrieve and analyse large quantities of apps automatically and provides a simple logic for other tools to integrate with. Finally, our software has been tested on a large set of real-world samples, both benign and malicious, demonstrating its efficiency (4s/app) and a reasonable capacity to evaluate the risk of Android app packages.

Keywords: Android application analysis, application risk level estimation, fuzzy logic algorithm

Received on 10 February 2014; accepted on 24 April 2015; published on 26 May 2015

doi:10.4108/ue.1.4.e5

1. Introduction

Nowadays, the vast majority (84.4%, Q3 2014, IDC) of smartphones [1] and a huge share (67.5%, Q1, 2014, IDC) of tablets [2] are powered by Android, a mobile operating system based on Linux kernel and maintained by Google. Every day, thousands of apps are published through the official or third-party application repositories. As shown in practice, many among this huge number of applications contain security and privacy risks [3], such as accessing the contacts, uploading location and retrieving device information.

These types of dangerous behaviour are common in both benign and malicious applications. Some are caused by developers' misjudgement, e.g. invoking suspicious adware or recycled codes. Some are caused deliberately to fulfil the requirements of application functionality.

The first reason is easy to understand, while to clarify the second one, we have the example of the instant message application *Viber*, which requires access to the user's contact list to find out who else is also using it. Moreover, as part of the authentication mechanism, it uses the mobile phone number as user identity. For this reason, during the account activation process, the *Viber* server sends a short message to the phone number with an activation code. Then the app installed on the user's device accesses the message and verifies the activation code to confirm that the user owns the phone number. Even if in this case there is an acceptable reason, accessing a contact list and SMS service is as equally risky as in malware. In scenarios like this one, the boundary between legitimate and malicious applications is blurred, and security-sensitive users may wisely avoid installing too invasive software.

The Android permission-based security model leaves the management of accessing control of device resources to end users. However, end users have almost no useful information about the danger of their choices, since the potential risk of an application is not evident. For this reason, we designed and implemented an automated Android app analyser, based on both static and dynamic analysis techniques, able to evaluate the potential risk level of an Android app package (*apk*). The analysis output consists of a detailed app behaviour report and a

*Corresponding author. Email: tao.su@polito.it

simple numerical value as comprehensive risk estimation, which can give a risk indication for both tech-savvy and common users prior to *apk* installation.

One of the biggest challenges in building fully automatic analysis systems is how to evaluate the analysis results in order to present end users a valid help for decision making. In fact, an automated system can successfully deal with objective truth but less easily with "reasonable" decisions. This phenomenon is also true in Android app analysis environment: in all previous researches, including static [4–10], dynamic [11–13] or hybrid [14] analysis approaches, this final decision is made by calling for human intervention. The analysis modules will filter out the majority of samples which do not trigger certain thresholds, then human inspection is required to categorise the rest samples. Although the filtering process can significantly reduce the human effort, it is still inconvenient for a market-scale analysis. Further researches address this point by applying complex reasoning techniques (e.g. machine learning, data mining, ...) [15–18] to make the automatic analysers capable of taking the final decision. We, on the contrary, adopt the fuzzy logic algorithm to overcome the uncertainty raised by the nature of automatic analysis. However, we do not claim the ability to directly detect malware, since, as a matter of fact, applications can be low-quality, buggy and risky without necessarily being malware.

This paper makes the following major contributions:

- we propose an automatic analysis approach exploiting both static and dynamic analysis techniques for Android app packages, and we map the detected activities to fine-grained risk categories;
- we evaluate application's risk level using the fuzzy logic algorithm, trying to overcome/mitigate the uncertainty limitation arose from the nature of automatic analysis;
- we implement a prototype, evaluating its effectiveness by analysing real-world benign and malicious Android apps, and we discuss the results and give an insight on the discriminating characteristics of the results for these two sets.

The rest of the paper is organised as follows: in Sec. 2 we present the Android security model, showing the basic mechanisms Android uses for protection, allowing readers to understand their limitations. In Sec. 3 we describe our analyser, including both the static and dynamic analysis modules, as well as the fuzzy logic system used in computing final results. After that, we present our evaluation results in Sec. 4, and in Sec. 5 we discuss previous work on Android application analysis and compare them with our analyser. Finally, in Sec. 6, we give a brief summary of our analyser and the results we achieved.

2. Android Security Model

Android operating system is based on the Linux kernel, and inherits many of its security features. For example, it takes advantage of the user-based permissions model to manage application execution; a unique Linux user identifier (UID) is assigned to every installed package. Consequently, applications are "sandboxed" in kernel-level and run as different users in separate processes.

Applications for Android are written in Java and run on a proprietary Virtual Machine called Dalvik (*DVM*). While the classic Java Virtual Machine is stack-based, Dalvik is instead register-based, which makes it faster on ARM microprocessors present on the majority of mobile devices. Java sources are compiled into *class* files using the Java Compiler (*javac*), and then converted into Dalvik bytecode (*dex* files) using the *dx* tool. The related resources (e.g., images and strings) are also compiled with the command `aapt` into a single file.

All the files are then packaged using *apkbuilder* into an *apk* (Android Package) file, which is, basically a *zip* compressed archive. This file is then signed with *jarsigner*, using a self-generated certificate. This certificate is checked by the underlying system only once, at the installation time.

Fundamentally, the components of an *apk* file are:

- a *META-INF* directory:
  - *MANIFEST.MF*, the manifest file containing the list of resources and their SHA1 digest;
  - *CERT.RSA*, the certificate of the developer;
  - *CERT.SF*, the SHA-1 digests of the resources in the *MANIFEST.MF* file.
- a *lib* directory, containing the compiled code specific to a software layer of a given processor, splits into more sub-directories, `armeabi`, `x86` or `mips`;
- a *res* directory, that contains raw resources which are not compiled into *resource.arsc*;
- an optional *assets* directory, that contains application assets that can be retrieved by *AssetManager*[1];
- *AndroidManifest.xml*, an additional Android manifest file, describing the name, version, access rights, referenced library files for the application. The file may be Android binary XML format that can be converted into human-readable plaintext XML format with other tools like *AXMLPrinter* [19];

[1]Provides access to an application's raw asset files.

- *classes.dex*, the source code of the application that is compiled in Dalvik executable format;
- *resources.arsc*, a file containing various types of pre-compiled resource, such as binary XML.

The manifest file contains the essential data needed by the operating system needs in order to install and run the application, including:

- the name of the Java code package, which can be used as a unique identifier for the application;
- the application components (activities, services, broadcast receivers and content providers);
- the list of permissions required by the application;
- the minimum/target level of the Android API (SDK version) required;
- the external libraries that the application must be dynamically linked to;
- the features of the device used (e.g., hardware sensors).

From the security point of view, the list of permissions required by the app and the used features (e.g., hardware sensors) are of particular interests. Sensitive APIs are intended for usage by trusted applications alone and are protected through the permissions security mechanism. Therefore, explicit permissions must be required a prior to installation, for example, the camera functions, location data, telephony functions, SMS/MMS functions and the network connections. Each permission is identified by a unique label, for example `android.permission.SEND_SMS`, `android.permission.INTERNET`, etc.. At installation time, the user is required to approve the permission list requested by the app, as shown in Fig. 1. If an application tries to use a feature whose permission is not granted, the system will throw an exception or return no results.

The permission-based model, while being intuitive for developers and users, presents some security flaws. As demostrated in practice, by itself it is not enough to prevent malicious software. For example, *TapLogger* [20] is a proof-of-concept key-logger which does not need any permission; it uses information from motion sensors of the device to deduce which keys the user has tapped. Another possible way to circumvent the Android security mechanism is by means of dynamic code loading. The code can be pre-stored inside the *apk* or even downloaded from the Internet at run-time, such code may contain malicious parts that are much more difficult to detect.

3. Android Application Evaluator

In mobile environment, either due to the software vulnerabilities or the users' tendency to allow more permissions than needed as well as the developers' inclination to ask for more than necessary, risky operations can easily cause privacy leaks or money losses for the device owner.

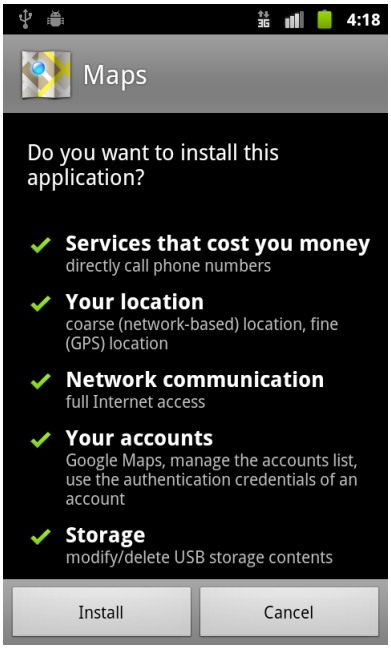

Figure 1. Permissions requested by an application.

To better inform users, our risk evaluator herein exploits various techniques, based on both static and dynamic analysis.

The goal of our system is to express, with a simple numerical value, the potential damage that the analysed app can cause to device and/or user; this value is called "risk score". The purpose of this value is to give a quick indication to the users, who can subsequently choose how to manage the potential source of threat, e.g., carefully read the detailed report provided by our tool.

In our context, the word "risk" is used for alerting about a tangible danger (e.g., a privilege escalation is signalled when a specific shell command is present, money risk when short messages are sent or phone calls are made, etc.). Conservatively, it also flags a potentially dangerous situation (e.g., the presence of a generic embedded binary executable), as well as potentially malicious behaviour patterns (e.g., read and then send the contact list). Each situation is mapped to a specific risk category according to different analysis patterns, as indicated in the following subsections.

3.1. Static Analysis

Static analysis is the process of analysing the source code of an application without actually executing it. Usually the starting point is decompiling the application's binary and generating a representation of the source code.

In Android, the first step is to unpackage the *apk* files (e.g., with a simple *unzip* command). The application manifest file (*AndroidManifest.xml*) is usually a key

source of information, and many tools exist to make it human readable (e.g. *AAPT*, the Android Asset Packaging Tool included in the Android SDK). By reading the manifest content, a number of tools can be employed to point out possible security issues. For example, Manitree [21], among a number of others, searches for the services shared with other apps without an intent filter or an explicit permission requirement, which would allow accessing from other apps. Meanwhile, since malware often sets higher priority values to forerun other app requests, the tool also looks at the intents' and the actions' priority values searching for insecure points.

In case the application has to be used in a trusted environment (e.g., on a device with sensitive data stored), a complete static analysis would require human inspections. To fully understand what an application does, the main file (*classes.dex*) has to be decompiled into human-readable codes. Different tools exist to dump Dalvik bytecode or to convert it to other low-level representations (e.g., Smali [22]). Generally, the result is easily understandable, unless obfuscation techniques have been used. Depending on the analysis goal, the preferred human-readable representation can differ. For example, an assembly-like representation, which is often easier to re-compile but harder to read, would be a better choice in order to modify and repackage the app.

In our case, the static analysis is implemented through several modules, which leverage extensively on the *androguard* APIs [23], an effective set of tools written in Python which helps implementing various static analysis methods on Android applications. We extended it through two complementary modules, *Behaviour* and *FileScan*.

Behaviour module. *Behaviour* is the first static analysis module, aims to check 1) whether the permissions required by the application are effectively used and 2) the critical APIs usage in order to finds out potential dangerous operations.

As the first step, *Behaviour* scans the app's manifest, retrieving the required permission list. Afterwards, it decompiles the app to obtain the source code, and the source code is analysed to find out what APIs are invoked and which operations the app attempts to execute. The tool can further check potential risks, e.g. privacy violations, frauds, device abuse and so on. In the final step, it correlates the APIs used with the requested permissions and detect incoherencies among them. A detailed list of all the behaviour patterns *Behaviour* identifies and their violated risk categories are shown in the appendix F.1.

In order to provide more details on the types of menace posed by dangerous operations, we enlarged the *androrisk* risk taxonomy and enable *Behaviour* module to map all the Android permissions [24] to an increased number of risk categories. The source code which lead to dangerous activities is mapped to the following risk categories:

- *root privileges escalation*, target app contains functionalities which require or exploit root privilege;
- *encrypted code*, target app uses crypto algorithms which can be used to obfuscate code;
- *binary code*, target app uses JNI, native code;
- *internet*, target app contains the Internet related activities;
- *dangerous*, target app calls dangerous APIs and permissions;
- *dynamic code loading*, target app calls for external libraries when executing;
- *exploit*, target app invokes functionalities which can be exploited (e.g., gingerbread exploit for Android 2.3 [25]);
- *phone*, target app contains functionalities which can affect the phone (e.g. enabling WIFI, accessing to phone settings);
- *SMS activities*, target app invokes API/functionalities which permits to handle SMS;
- *money*, target app contains activities that costs phone owner's money (e.g., phone calls, sending SMS);
- *signature*, target app declares *Signature* permission in the *manifest*[2];
- *signature system*, target app declares *Signature-ORSystem* permission in the *manifest*[3];
- *privacy violation*, target app contains activities which violate user's privacy (e.g., accessing to contacts, GPS location).

The rationale behind this mapping is to enumerate and characterise the possible danger the user might face, and present this result in a way which is immediately meaningful to the users. Some of the

[2] *Signature* permission is a permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.

[3] *SignatureOrSystem* permission is a permission that the system grants only to applications that are in the Android system image or that are signed with the same certificate as the application that declared the permission. So this permission should never be granted to third party developers besides the ones that publish the OS image. This permission should be used only in very special cases when multiple vendors have applications built into a system image and need to share specific features explicitly because they have to be built together. In general, this and the previous permission should not be used by 'common' applications. Since when the OS grant them, the application can do some critical operations, even without user notification.

mappings are straightforward (e.g., the *root privileges escalation* activity is categorised into the homologous risk category). Some other activities are mapped into multiple risk categories; for example, the *Internet* activities are categorised into both *Internet* and *Money* risk categories. In this way, we can evaluate the app's dangerousness based on both the detected activities and on a finer-grained level of risk categories and their violation occurrences, which are more understandable by end users. Thanks to this more accurate categorisation, the evaluator can compute the risk score which accurately reflects the app's behaviour.

FileScan module. *FileScan* is the second static analysis module, which analyses every file stored inside the app package. It is structured as a Python class which can recursively analyse *apk* files, as well as *zip*, *tar*, *gzip* and *rar* archives. After analysing every file stored inside the archive, it computes a risk score which represents the potential harm posed by that *apk*. It can provide various information about the *apk* file (e.g., files list, checksums, URLs and phone numbers found inside them), point out suspicious behaviour patterns (e.g., hidden files, shell scripts with potentially dangerous commands) and identify known infected files (e.g., embedded malware *apk*, infected native libraries).

FileScan accomplishes all these task by means of a recursive approach, it automatically analyses every single file and identifies its type using the magic number analysis. Magic number analysis uses the magic number (a 2-byte identifier at the beginning of the file), as well as specific patterns, in order to identify the format of a file without relying on its extension. The output of this analysis is the MIME type of the file, a standard two-parts identifier, which is used by *FileScan* to correctly categorise the file.

Many malware apps attempt to conceal their purposes, and often alter file names to use some innocuous extension (e.g. *png*) and deceive anyone who would quickly analyse the app content; for example the malware families of DroidDream and GingerMaster use this trivial technique. *FileScan* can identify the dangerous files even if they have been renamed; moreover, it considers the case of a renamed critical file (embedded application or binary) as a clear sign of malicious intention. In order to decide whether the extension of a file has been changed, two approaches are used: comparing the extension against a list of valid extensions (white list approach) and against a list of invalid ones (black list approach).

Embedded apps are *apk* files stored inside another app package, that can be installed or loaded at run-time through dynamic code injection. Many static analysis systems are not able to detect and analyse them properly; some malware use this technique to carry another application with other functionalities. The *elf binaries*, either executables or shared libraries, can be used by the app for a direct access to the system APIs. Usually they are used for performance or compatibility reasons, and they can be called from the main application as external libraries. Such code is more difficult to be detected and analysed; moreover, since the system call interface of Linux kernel is directly exposed to the native code, it can be used to exploit the system vulnerabilities. *Shell script* files are textual files containing commands, which our module can identify as threats; for example, they can be used to perform privilege escalation attacks.

In order to determine if a file is malicious, *FileScan* can look up checksums of the files using the Malware Hash Registry online service[4], and retrieving the detection rate for that file. It also uses a limited set of checksums associated with infected binaries from known families of malware, in order to speed up the analysis process and be able to identify malicious file even when the online lookup service is not available.

Concisely, the risk categories considered and estimated by *FileScan* are the followings:

- hidden elf binary, analysed apk archive contains an ELF file which is not in the standard directory and has unexpected extension;
- hidden *apk*, the archive contains an embedded *apk* file with an unexpected extension;
- hidden text, the archive has a textual file with an invalid extension;
- infected elf binary, the archive has an ELF detected as infected;
- infected *dex* code, the archive contains a *dex* file which is detected as infected;
- input shell, the archive contains a shell script;
- shell install, the archive contains a shell script with install commands;
- shell privilege, the archive has a shell script containing commands usable to perform a privilege escalation;
- shell other, the archive has a shell script containing dangerous shell commands.

FileScan is also able to look for URLs and phone numbers inside textual files, which could be used by the app to communicate with malicious C&C (command & control) servers, to make phone calls or send short messages to. URLs and phone numbers are also searched in the string dictionary in the application package, which is contained in the compiled resource file (*resources.arsc*). The regular expression used for URL addresses is able to identify URLs with escape characters or formatted parameters, which could be manipulated by the application to produce valid addresses. Moreover, a

[4] www.team-cymru.org/MHR.html

white list of the most common URLs is used to filter out irrelevant results. The regular expression used for phone numbers is able to find potential phone numbers composed by 4 or more digits, but the false positive rate in this case is significant and manual controls are needed. However, this shortcoming can be mitigated by combining the results from *FileScan* with the analysis result of the dynamic analysis module, which is able to identify phone numbers and URL addresses used during the sample's execution.

FileScan is, to the best of our knowledge, the first tool capable to automatically analyse all the files in Android app package and detect these kinds of menace. Although it is not able to defeat more advanced techniques (e.g., file encryption), it can quickly and efficiently identify a wide set of dangerous alternatives. In our opinion, at the moment it achieves the best result for an automated analysis of this type.

3.2. Dynamic Analysis

Dynamic analysis is the run-time analysis of applications, performed by executing the samples inside a controlled environment. The environment should be instrumented to collect various types of information during the execution, which can be used in a real environment or in an emulated one. Emulation is the cheaper solution, but it suffers some limitations. For example, the emulated environment might not connect to the real communication network and some specific firmwares cannot be satisfactorily emulated. The obvious advantage of a real environment is the accuracy of the answers and the connection to the real world, but it is much more complex and expensive to manage in a secure way. For the sake of reproducibility, we chose the approach of emulation.

Our dynamic analysis module is developed on top of *Droidbox* [26], a well-known open-source dynamic analysis tool for Android applications. *Droidbox* lies in the security analysis suite category. It is an open-sourced sandbox for Android apps [26], which uses tainted data tracking and function call monitoring techniques. It is developed by Patrik Lantz using Python programming language. The following information will be shown when the analysis ends.

- digest of the analysis package;
- network operations;
- file system accesses;
- services started and classes loaded through DexClassLoader;
- cryptographic operations performed using Android API;
- tainted data which leave the system through network, file or SMS;
- broadcast receivers;
- SMS(es) sent and phone number(s) called.

Even though the original version of *Droidbox* can provide a comprehensive analysis result of the apps, it is quite inefficient to import and generate output. Our module enriches *Droidbox* in a number of ways, especially from the input and output data processing points of view. The modified version can input the selected apps continuously from a set of samples, and create a clean virtual device image for each of them. In order to simplify the work for further analysis, we extended the tool's output, such that all detected activities and relevant information (e.g. phone numbers, URLs and file names used by the sample) are stored in separated files. In this way, dynamic analysis can be totally automatic to analyse multiple number of samples; this is, to the best of our knowledge, very rare in dynamic analysis systems.

Analogous to static analysis modules, the activities detected by dynamic analysis module are mapped into the following risk categories, to provide a finer-grained basis for the fuzzy evaluation system.

- *encrypted code*, target app uses Android encryption APIs;
- *binary code*, target app invokes `dex` class loader, that may execute external code;
- *dynamic code loading*, target app calls system's native functions;
- *exploit risk*, target app contains and runs an exploit;
- *internet*, target app uses Internet service;
- *money risk*, target app executes money cost operations;
- *SMS activities*, target app accesses or sends SMS;
- *privacy violation*, target app collects private data (e.g., device ID, contacts, IMEI);
- *phone abuse*, target app harms the system's integrity.

The detailed mappings between detected activities and their risk categories are shown in the appendix F.2.

Our automatic dynamic analyser is very effective against risky apps which execute dangerous operations directly after they are installed and started by *adb*. If some stealth techniques are used, for example a hidden trigger, our analyser, as most automatic systems, needs human interactions to trigger the dangerous operations. Moreover, new generation malware is capable to find out whether it is running inside an emulator [27], more advanced malware can even detect whether it is running inside a device belonging to a security analyst for research purposes (which usually has stored no personal data inside). Nevertheless, in most cases, the risk scores our analyser computes can offer realistic and reliable danger level estimations from a fully automatic analysis point of view.

Another drawback of the dynamic analysis consists in its time-consuming nature. The emulator needs to start up a clean Android virtual device image for each *apk*, and then it has to wait for the tested app to finish all its initial operations. In average, in order to obtain reasonable results, a complete analysis of a single sample should take up to 5 minutes and no less than 3 minutes. Therefore, for the sake of efficiency, in many cases dynamic analysis is only performed on *apks* which are classified as "risky" by the static analysis modules (i.e., the static analysis risk score is situated above a threshold), in order to confirm the dangerousness of the sample.

3.3. Applications Risk Evaluation

The fuzzy logic is widely used in decision making systems. As stated by Prof. Zadeh in [28], "*fuzzy logic is a precise logic of imprecision and approximate reasoning*". It is capable to converse, reason and make rational decisions in an environment of imprecision, uncertainty, incomplete information, conflicting information, partiality of truth and partiality of possibility, which is exactly the case of Android application risk level estimation.

However, the fuzzy logic is not the only option, and other scoring algorithms can also be adopted. As a matter of fact, we keep the analysis modules and evaluation system separate intentionally, to facilitate further experiments with alternative scoring algorithms.

In spite of the scoring algorithm, the risky activities detected and their mappings to risk categories remain most valuable outputs, which allow end users a fine-grained inspection of the application's characteristics.

Based on the fuzzy logic rules, the evaluator outputs a *risk score*, which gives a quick indication to the users about how dangerous the app may be, so that they can give permission informed about the potential sources of threat. It should be noted that, in our context, the word "risk" is used for alerting about a tangible danger, not necessarily the presence of malware.

To make the evaluator as flexible as possible, the fuzzy logic risk scoring system is embedded alongside the three analysis modules as indicated before, so that the modules can be used independently (to have a quick feedback) or together in cascade (to have a fully detailed insight).

Fuzzy interpretation of risk states. The input of the fuzzy logic system are derived from the risk categories and their corresponding violation frequencies in each module. For each risk category, the dangerousness level to end users is not equal. For instance, the risk categories associated to money and privacy are considered the most dangerous ones as they are the biggest concerns to end users. Therefore, we defined four separated states domains for each risk category, from the least to the most dangerous estimation, they are `LOW`, `AVERAGE`, `HIGH` and `UNACCEPTABLE` risk states. Linguistic logic is used since human understandability matters to the end users while it can be easily interpreted using the fuzzy logic.

Fuzzy sets assign a truth-value called `probability` in the range [0,1] to each possible value of the domain. These values form a `possibility` distribution over a continuous or discrete space. The violation occurrences combined with truth-value of each risk category determine its state. If we consider the violation occurrence as a discrete space $[0, +\infty)$ with increment equals to one, then we can present graphically the possibility distribution of the state given the risk category.

As an example: the risk states associated to `BINARY_RISK` in *Behaviour* module is defined below and shown in Fig. 2.

- Definitely LOW from 0 to 6, and not LOW if higher than 10;
- Not AVERAGE if lower than 6, AVERAGE at 10 and not AVERAGE if higher than 15;
- Not HIGH if lower than 10, HIGH at 20 and not HIGH if higher than 24;
- Not UNACCEPTABLE if lower than 23, and absolutely UNACCEPTABLE if higher than 30.

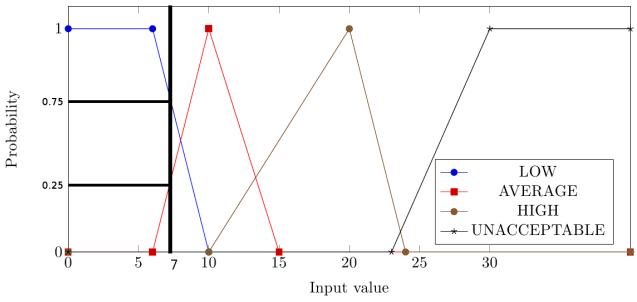

Figure 2. Adjectives defined for `BINARY_RISK`.

For instance, if the violation occurrence is 7, then `BINARY_RISK` is 75% in `LOW` risk state and 25% in `AVERAGE` state.

The scoring system can be tuned to better adapt to specific context (e.g., security sensitive environments). For example, to give more weight to `BINARY_RISK`, the adjectives for each risk state can be reduced hence `BINARY_RISK` reaches `UNACCEPTABLE` state with less violation occurrences.

Selecting the boundaries for these adjectives is challenging. The expected outcome is the realistic risk level of analysed samples; therefore we needed to improve our experience to achieve this result. The tool requires iterative tuning, so that the most relevant risk categories (e.g., `MONEY`, `PRIVACY`) weight more than others (e.g. `INTERNET`), until the final result is meaningful for the end users.

At the end of this step, each sample should have a set of risk states for all categories.

However, *filescan*, requires different settings. Since it looks for the most peculiar behaviour of the *apk* archive (as a matter of facts, they are most likely to be malicious behaviour), we set the risk categories differently than other modules. The risk states are set to be "HIGH" once *filescan* detects a corresponding event, so its tolerance is stricter; therefore the results from *filescan* have more influence compared with the other two modules.

Computing fuzzy risk level. To combine the states of all risk categories in the scoring system, the fuzzy logic rules are required. However, before defining these rules, the output of the rules were defined and their adjectives were associated using `singleton` functions to simplify the computation as following:

- NULL_RISK to Singleton(0.0);
- AVERAGE_RISK to Singleton(30.0);
- HIGH_RISK to Singleton(70.0);
- UNACCEPTABLE_RISK to Singleton(100.0).

Defining the fuzzy logic rules that associate the fuzzified input variables (i.e. the risk state set) to the output adjectives is a key domain in influencing the final result. In the current configuration, the system is governed by more than 100 rules aggregated in these three analysis modules. All the rules will be evaluated, and if true they will contribute to the final risk score.

To give a very simple example with parameters defined in Fig. 2, if a rule states:

`IF BINARY_RISK IS` AVERAGE `THEN output IS` HIGH_RISK

In the case that the violation occurrence is 7, and this is the only rule in the scoring system, the risk score will be:

$$(truth\_level) * (adjective) = 0.25 * 70 = 17.5$$

If, the only rule in the system is changed to following:

`IF BINARY_RISK IS` LOW `THEN output IS` AVERAGE_RISK

then with the same input value, the output risk score will be:

$$(truth\_level) * (adjective) = 0.75 * 30 = 22.5$$

The last step of computing the risk score is called *defuzzification*, which can be performed using several different methods. In our case, the fuzzy logic systems in all three modules use the `Centroid Method`, which means to calculate the centre of gravity for the area under the curve. Thanks to the choice of `singleton` function, this computation is simple to understand. The formula is the following:

$$COG = \frac{\sum_{x=a}^{b} u_A(\chi)x}{\sum_{x=a}^{b} u_A(\chi)}$$

Variables a and b represent the attributes in the fuzzy logic system, from `NULL_RISK` to `UNACCEPTABLE_RISK`. While $u_A(\chi)$ indicates the `truth level` for all the attributes, and x is the adjectives for each attribute.

As an example, the final risk score with only two rules defined before and input value equals to 7, is computed as:

$$Risk\_score = \frac{(0.25 * 70) + (0.75 * 30)}{0.25 + 0.75} = 40.0$$

Of course, there are rules with more complex conditions in our fuzzy logic system. They combine multiple risk categories using logical operators like *AND*, *OR* and *NOT*, which will highlight some specific dangerous operations treated as heuristics. For example in dynamic analysis module, leaking data to the Internet operation will violate `PHONE_STATE_RISK` and `INTERNET_RISK`. Hence, if both risk categories are at `HIGH` risk state, then the final risk score should be significantly increased. Similarly, for other obvious dangerous actions, there are corresponding rules to leverage the final score. On the contrary, if certain risk category combinations are in `LOW` state, the final risk score will decrease.

The rules with `FALSE` condition will give no contribution. Otherwise, the rule's output will concur to the final risk score. Hence, apps with less obvious dangerous operations will have smaller risk scores than the ones with more obvious dangerous operations. Even though in some cases, less risky apps may have more violation occurrence in certain risk category. Thus, the result is not monotonic solely based on the occurrences but leverage more on the heuristics, which gives more accurate indications of application's risk level. The same type of heuristics is applied in all three modules. For example in *Behaviour* module, if the application has permissions to access user's contact and in the mean time has the permission to send data through the Internet, the final risk score is higher than the one only has the permission to send data through the Internet. Even in some case, the latter can send data more times than the former when the behaviour is confirmed using the dynamic analysis module.

4. Experimental results

To perform an extensive testing of the system, we developed an additional software module, the *AppsDownloader*, which is based on the unofficial open source project named Android Market API [29]; it can automatically retrieve free apps from any Android repositories and also from the local file system.

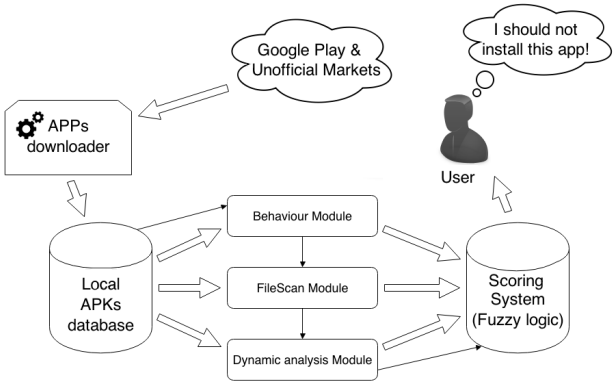

Figure 3. Danger level evaluator testing architecture.

Exploiting the AppsDownloader and the workflow described in Fig. 3, we tested our analyser against a set of 41000 free goodware applications (this set will be referred as *market set*); and a set of 1488 known malware samples from 90 distinct families, from the *Android Genome Project* [30] and *ContagioMiniDump* [31] (this set will be referred as *malware set*).

The number of goodware is significantly higher than the number of known malware. This follows the real world situation: there are magnitudes more goodware developers than malware developers. Meanwhile, even though the number of goodware sample is imbalanced with the number of malware sample, this huge number of application samples can be used to show the efficiency of our evaluator in a meaningful way. Thanks to Android Genome Project [30], the malware dataset we used in our experiments is one of the largest compared with previous work.

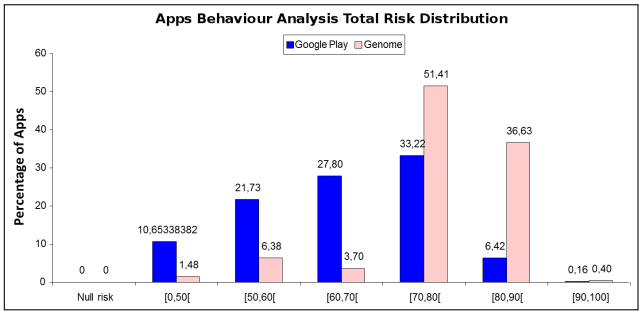

Figure 4. *Behaviour* module results: app risk scores distribution, market vs. malware.

In the first place, as shown in Fig. 4, 40% of the *market apks* obtained a risk score greater than 70 tested using *Behaviour* module, while 88% of *malware apks* obtained a risk score greater than 70. The result is in accordance with Felt's result [5], which shows one-third of apps in Google Play are over-permissioned. For this reason, the discrimination power of *Behaviour* is limited. However, risk score is only for indicating risk level, thus apps

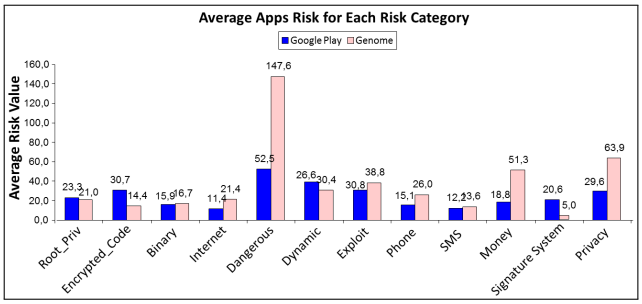

Figure 5. *Behaviour* module results: average app risk value per category, market vs. malware.

exceed this threshold will be considered risky in the case of permission abuse and calling potentially dangerous APIs. The threshold is set to be 70 is not the final settlement, it is only a choice based on our experiences. If a stricter criteria is needed, this threshold can be reduced to 60. But in this case, more application samples need to analysed dynamically, and much more time and resources are needed.

From the risk score distribution, we can also see that the scores of free applications are concentrated in the interval from 60 to 80, while the scores of known malware are in the intervals from 70 to 90. The histogram from Fig. 5 shows the distribution of average app risk scores for each risk category in both *market* and *malware* sets. The distribution patterns are contrasting. Known malware has conspicuous peaks on `Dangerous API`, `Money` and `Privacy` risk categories, while *market* apps have a smoother distribution.

To reason about the limited discrimination power of our *Behaviour* module, we studied in more depth about the distinguishing characteristics of the malware and the goodware dataset. The result shows that the patterns of goodware and malware referring to the requested permissions and the identified behaviour is quite similar.

As shown in Fig. 6, the five most common permissions required by the goodware are:

- `INTERNET`;
- `ACCESS_NETWORK_STATE`
- `READ_PHONE_STATE`
- `WRITE_EXTERNAL_STORAGE`
- `READ_CONTACTS`

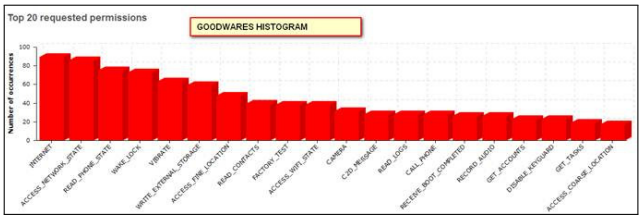

Figure 6. The top 20 required permissions of goodware.

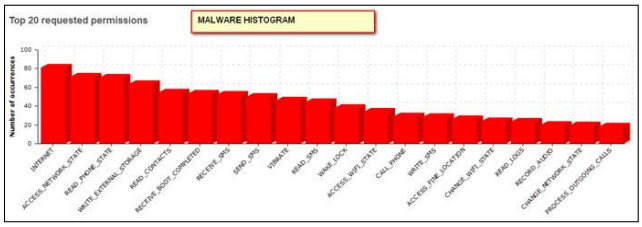

Figure 7. The top 20 required permissions of malware.

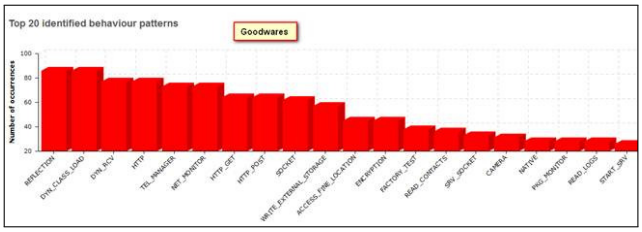

Figure 8. The top 20 identified behaviours of goodware.

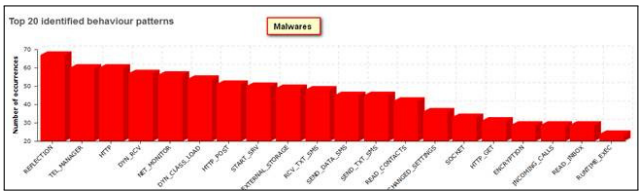

Figure 9. The top 20 identified behaviours of malware.

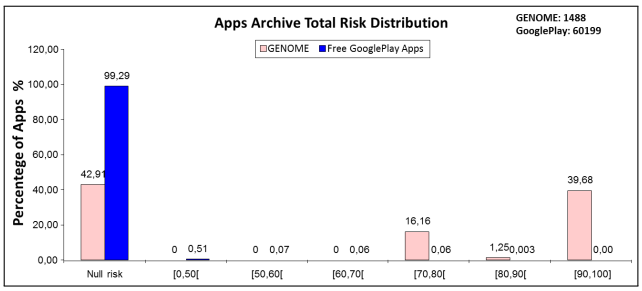

Figure 10. *FileScan* module results: app risk scores distribution, market vs. malware.

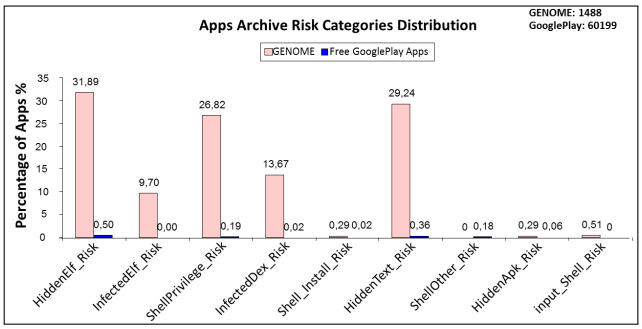

Figure 11. *FileScan* module results: app percentage per violating risk category, market vs. malware.

Three of these permissions are also the most requested ones by known malware as illustrated in Fig. 7. For this reason, we presume and confirm that, it is quite hard to find the differences between goodware and malware by only examining the requested permission list. Some research uses the permission combination to indicate the potential privacy risk level, such as [32]. However, by solely relying on permissions and their combination patterns, it is extremely difficult (sometimes just not possible) to discriminate between malware and goodware.

Further results on possible differences between the behaviour of goodware and malware samples are illustrated in Fig. 8 and Fig. 9. Of the top five identified behaviour patterns, four of them are the same for the two datasets. They are:

- `REFLECTION`, java reflection, makes it possible to inspect classes, interfaces, fields and methods at run-time, without knowing the name of the class, methods, etc.;
- `DYN_RCV`, app is loading one or more *receivers*[5] dynamically, without declaring them in the manifest file;
- `HTTP`, app is trying to issue a HTTP connection;
- `TEL_MANAGER`, app is trying to get telephony service information on the device.

The most interesting point here is that, goodware tends to use `DYN_CLASS_LOAD` during the execution, which can invoke code from shared libraries or even external parties, putting user's device in danger. The figures are another evidence that solely rely on behaviour patterns is not enough to discriminate goodware from malware, as in the same case of permissions.

After *Behaviour* analysis, *FileScan* module tested both sets. The result is shown in Fig. 10, 99% of the *market apks* has a null risk score, while 40% of the *malware apks* has a risk score greater than 80. The distribution of applications on *FileScan* risks categories is shown in Fig. 11, we can see that the `HiddenElf`, `ShellPrivilege` and `HiddenText` are the most violated risk categories by known malware. In our dataset, the samples with peaking risk score (i.e., 100) are mostly from `GingerMaster`, which contains *shell install* commands inside the package.

Dynamic analysis works as a confirmation mechanism, to prove the dangerousness of suspicious apps. To the application samples which have static analysis risk scores above the threshold (70 for *Behaviour*, and non-zero for *FileScan*), they are subsequently analysed by the *dynamic analysis* module. The results are shown in

[5]receivers enable applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running.

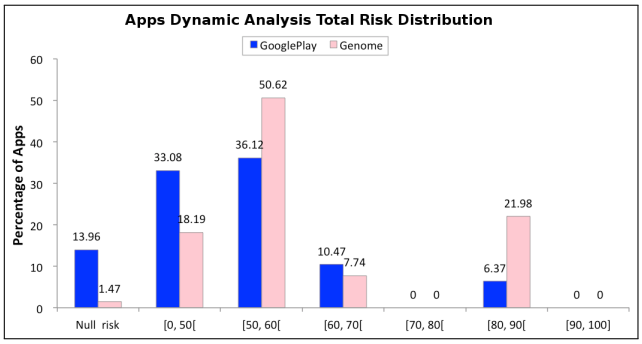

Figure 12. *Dynamic analysis* module result: app risk scores distribution, market vs. malware.

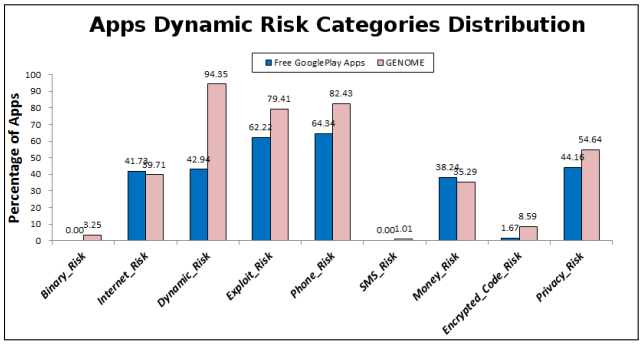

Figure 13. *Dynamic analysis* module result: app violation percentage per risk category, market vs. malware.

Fig. 12 and Fig. 13: 50% of market *apks* obtained a risk score greater than 50, while 80% of malware *apks* exceed the same threshold. Since the analysed samples are labelled as "risky" application before they are analysed by the dynamic analysis module, we set a smaller value as the threshold to alert the end users. As shown in Fig. 13, almost all known malware violate the `DYNAMIC` risk, which is derived from the use of system's native functions, such as dynamic code loading. This suggests that dynamic code loading needs stricter control in the future. Only known malware exploit the activities related to the `BINARY` (`BaseBridge` samples) and the `SMS` (`HippoSMS` and `FakePlayer` samples) risk. Moreover, the phone numbers, URL addresses and the activities detected during execution are valuable, since they indicate the real behaviour of the analysed samples after their installation and initiation.

No final combined result of all three analysis modules will be presented, since there is no obvious way to put weight to these three results: all three analysis modules have their scopes of analysis, and need the others to balance theirs result. For this reason, combining the individual result of these three modules is not a good choice. A much better approach is to extensively leverage on the fuzzy logic algorithm, before combining the outputs of each analysis modules, these outputs can be pre-configured to be mapped into the risk categories as a whole set. Thus, there will be only one fuzzy logic evaluator to evaluate all the outputs of three modules as a whole. However, in this way, since the fuzzy logic evaluator requires the output of all three modules, the flexibility of the analyser is removed. As a trade-off, we only provide the individual results of the analysis module, and do not combine them as a final result.

From an overall perspective, the analysis of the *market apks* gave an unexpected high risk values on `Internet`, `exploit`, `phone` and `dangerous APIs` risk categories, while the *malware* set gave high values on the others, like `money`, `dynamic`, and `privacy` risk categories, and the significant risks on archive files (`HiddenElf`, `ShellPrivilege` and `HiddenText`). As far as *malware* set is concerned, `user privacy` violation is the most important risk category encountered, whilst, for *market apks*, the `dangerous APIs` risk category is the highest.

Besides the risk scores, the analysis system exposes and confirms 288 suspect URL addresses and 5 phone numbers identified in the tested applications. We found that the most frequent URL addresses detected are PayPal websites which provide payment services, and often among them we can find collectors of the leaked information. Regarding the mis-usage of SMS and phone calls, only certain known malware tries these unauthorised communications, since they are pretty easy to be detected by the users.

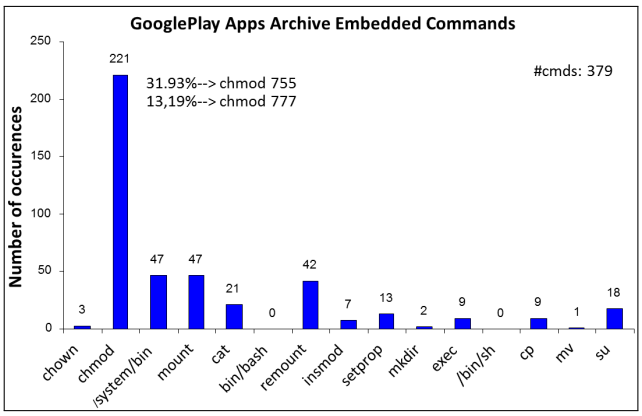

Figure 14. embedded commands in market apps.

We also investigated the most common shell commands encountered in the apps from both the *market* and the *malware* datasets. The results are presented in Fig. 14 and Fig. 15. Since the number of goodware samples is magnitude more than the number of malware samples, there are more commands detected in the goodware samples. We detected 379 different commands with different arguments in all these 41000 free market applications. `chmod` has been detected 221 times, and of 31.93% (71) times it is `chmod 755` and of 13.19% (29) times it is `chmod 777`. While in the malware dataset,

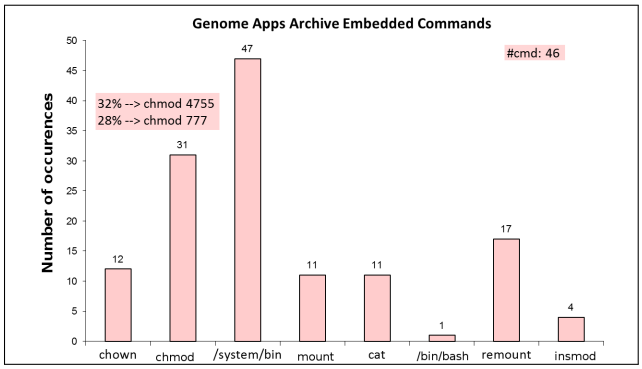

Figure 15. embedded commands in malware apps.

we only detected 46 different commands with different arguments. On the contrary to goodware dataset, the most detected command is `/system/bin`; it has been detected 47 times. Although only a limited number of analysed apps contain *FileScan* targeted activities, the results show that, practically, there is no difference between the most recurrent commands identified in both sets.

A significant result of our tests is that the *market apks* could not be as innocuous as the users are inclined to think. This may be an effect of a poor programming or presence of potentially unwanted codes (mainly due to adware or recycled code), but the risk is there, even for apps that users may be tempted to trust. The main critics to developers is the permission abuse (i.e. permissions requested but not used), the presence of recycled dangerous code and the use of dangerous APIs, where alternatives do exist. All these weaknesses transform *apks* into attractive and powerful targets for *trojans*[6], which can exploit these over-permissioned *apks* to unrestricted acts on end user devices.

5. Related work

Our analyser aims to inspect a market-scale number of applications without any human interaction, and tries to overcome the uncertainty that arises from the nature of automatic analysis. Its goal is to highlight the risk level of Android applications but not to directly detect malware.

As stated in Sec. 2, Android relies on permissions to limit the apps' functionality. The principle of "least privilege" is recommended and suggests that an application requests only the most restrictive set of permissions for performing the task at hand. Unfortunately, this principle is seldom respected, because of either Android's disorganised documents on `Permissions`[7] or developers' tendencies to require more than needed, which could easily bring risks for the users. Thus, many methodologies available to check the requested permission in search of risks of misbehaviour. In [33], the authors use probabilistic generative models for evaluating the potential risk of analysed applications. However, this evaluation system relies solely on the number of permissions required by the sample and gives monotonic results. Another tool in this category is *Stowaway* [5]; it identifies app permission abuses, by mapping the permissions required in the manifest to the invoked APIs, and detects the incoherencies between them. In their experiments, one-third of the apps were found to be overprivileged. A more effective approach is proposed in [34], the authors evaluate app risks on the basis of how rarely permissions are required for apps in a specific category, like navigating or games. However, since Android's permission model fails to fully control application behaviour, analysing solely the permissions requested can only be a starting point, but it is incomplete for a reliable evaluation of the application's risks.

Other non-permission-based static analysis approaches are also proposed.

Taint analysis addresses the problem of analysing Android apps based on their data flows. In [6], the authors propose *FLOWDROID*, a novel and highly precise static taint analysis for Android apps. With the help of Android-specific challenges like the application life-cycle or callback methods, *FLOWDROID* can give more concrete results about the data leakage. *CHEX* [7], *AndroidLeaks* [8], *LeakMiner* [9] all use the same static taint analysis approach to analyse data leakage caused by Android apps.

Inter-component communication (ICC) is also a studying point to analyse Android apps. In [35], the authors recast ICC analysis to infer the locations and substance of all inter- and intra-app communication in an Android environment. In this way, it can detect dangerous communications between applications and identify new types of the risky operations such as transitive privilege usage [36]. Similarly, *ComDroid* [10] also attempts to identify security risk in Android apps by analysing inter-application communications. However, the same as in the permission-based approach, the analysis results using previous methods can only cover partially the surface of Android application risk analysis.

One important component in applications is advertisement libraries, especially for free applications. Many developers include such libraries to obtain some remuneration for their effort, but few of them fully understand the risk implication or fully control their behaviour.

[6] A Trojan horse, or Trojan, in computing is a generally non-self-replicating type of malware program containing malicious code that, when executed, carries out actions determined by the nature of the Trojan, typically causing loss or theft of data, and possible system harm.

[7] `http://developer.android.com/guide/topics/security/permissions.html`

AdRisk [37] analyses in-app advertisement libraries, and systematically identifies the potential risks. The results show these libraries may also contain potential dangerous operations ranging from leaking user's private information to executing untrusted code. We suspect that, it is one of the reasons that benign apps have unexpected high risk scores.

RiskRanker [4], among others, has a broader coverage. It exploits a proactive scheme that requires no malware specimen and their signatures. It provides two orders of risk analysis, firstly by statically analysing whether sample exploits platform-level vulnerabilities, and secondly searching for specific behaviour patterns, which malware commonly adopt but that is uncommon among legitimate apps. The result shows that *RiskRanker* is quite efficient to detect zero-day malware. But the detection mechanism can be easily circumvented by informed malware developers. Our work shares the same goal with *RiskRanker*, to identify the application risk in advance but, our analyser provides a broader coverage. Our static analyser extends this approach with the *FileScan* module, which analyses all the files stored in *apks*. It is able to pin point dangerous and potentially malicious files, such as embedded apps or hidden commands. In this way, by combining the analysis performed by *Behaviour* and *FileScan* modules, our static analyser strives to cover a larger surface and gives more concrete results of the risk level of analysed samples. Furthermore, our dynamic analysis module provides a thorough analysis of the analysed app running in an emulated environment, showing the real behaviour of the suspicious samples, and possibly confirming its potential risks.

DroidRanger [14] uses a permission-based behavioural footprinting and heuristics-based filtering to analyse Android applications, and call for dynamic monitor to detect the maliciousness. The analysis result, supported by human inspection, shows this system is effective for both known and zero-day malware detection. Our analyser works with a similar approach, using the static analysis to highlight suspicious apps, and the dynamic analysis module to confirm the dangerousness. Yet, we have different purposes. *DroidRanger* aims to detect malware in the official and third-party markets, with maliciousness confirmed by human experts. On the contrary, our analyser aims to evaluate application risk entirely without human inspection, while the final decision is made by the fuzzy logic scoring algorithm.

Dynamic analysis techniques follow another path. TaintDroid [11] exploits taint analysis on data flow in an emulated environment. It is still the state of the art taint tracking system for Android. It taints sensitive data and tracks them in the operating system, and gives alerts when they leave the device at taint sinks. However, it has significant false positive rate when tracked data contain configuration identifiers. Moreover, the native library loader used in the image has to be modified so that applications can only execute in user-space and with native system libraries.

DroidScope [12] supports virtualisation-based malware analysis, and provides both OS-level and Java-level semantics. On top of *DroidScope*, the authors develop several analysis tools to collect behavioural information.

VetDroid [13], on the contrary, reconstructs app's behaviour with permission use analysis. Dynamic analysis requires a significant amount of time, usually no less than 2 minutes for a complete run. Moreover, the false positive and negative rates are relatively high. Furthermore, it is hard to be automated and to detect hidden triggered operations. Thus, the information collected is most likely to be incomplete. Hence it is advisable to be used as a confirmation mechanism instead of a stand-alone evaluation tool, as what we have in our analyser.

In order to process market-scale apps, a fully automated analyser is required, however using retrieved information to make clever and automatic decisions is a challenging task. In previous works, machine learning techniques have been introduced to overcome this problem. In this context, *MAST* [15] uses Multiple Correspondence Analysis (MCA) technique to measure the correlation between the declared indicators of functionalities to be presented in app's package. It needs a large dataset (including both benign and malware samples) as the training data, then applies the correlations to the analysed samples. Similarly, in [16], the authors apply pattern mining technique to permission request patterns of Android apps. They discover the correlation between applications' permission request pattern and their belonging categories. Furthermore, they devise low-reputation apps often deviate from the pattern identified from high-reputation apps. Using machine learning technique to make the final decision is promising, but it requires a huge amount of preparation to fetch a training set with necessarily large number of applications. On the contrary, using the fuzzy logic algorithm is simpler and straightforward. Also the cost is fair; each computation takes only around four seconds. Although parameters need to be tuned to improve accuracy, the results can still give acceptable indication of analysed sample's risk level.

6. Conclusions

In this paper we presented a combined static and dynamic analysis tool for Android application risk evaluation. Its purpose is to effectively evaluate the risk level of an application (whether it is malware or not), to inform the user decision to use it or not.

The analysis system is based on the software modules *Behaviour* and *FileScan* for the static analysis of the code and archive files, and on our improved

version of *DroidBox* for the dynamic analysis as a confirmation mechanism. In our opinion, our tool has good extensibility as it can be easily used to integrate with other analysis tools, for example, the one to detect repackaged apps. In face, according to the figures from [38], 5% to 13% of apps in the third-party markets are repackaged, and it is highly likely that a lot of them are of poor quality; and, of course repackaged apps are the favourite disguise for malware.

Our system can execute static and dynamic analysis separately or in cascade. This allows flexibility in a number of scenarios. If time is limited, the dynamic analysis can be performed only on the *apks* labelled as potentially harmful by static analysis. If a more accurate check is required, both can be conducted to provide a complete report on the application.

Finally, we performed a detailed analysis on a statistically significant dataset, containing more than forty thousand applications, to test the efficiency of the system. The tests highlighted the capability of our analyser to evaluate the risk level of Android applications. Furthermore, since malware and goodware have been categorised according to a set of risk parameters (derived from the Androguard taxonomy), our system gives a statistically sound insight into present app risk characteristics. On the one hand, this can help short term strategies to contrast malware diffusion. On the other, it highlights excessive amounts of required permissions from app developers, and is a flag to demand more security-aware application development guidelines.

Future developments should include improvements to risk indication reliability and understandability, experimenting our methodology with different risk evaluation algorithms, and presenting a customisable set of risk indicators on the basis of the specific end-user characteristics.

However, the first step is to improve the accuracy of the analyser. There are two possible solutions: the first one is to provide a finer-grained risk category set, generating risk categories with more evident malicious behaviours. For example, in the dynamic analysis module, a new risk category called "SMS_TO_PREMIUM_NUMBER" could be introduced, indicating that the analysed app will send a short message to one of the known premium numbers. The same approach can work with other analysis modules. The second solution is to integrate with more analysis modules. For example, *MalloDroid* [39], which is capable of identifying broken SSL certificates in an app's code [40], is in the to-be-integrated-with list. In the mean time, malware like *hoser* [41] with the capability of obfuscating its behaviour by means of obfuscation/packaging tools seems likely to be a popular trend. To deal with this problem, we plan to introduce an unpacker tool (e.g. [42]) to improve the accuracy of our analyser.

Meanwhile, another relevant improvement could be to integrate the application risk level evaluator with a third-party store. In order to indicate the risk level of the application and to help users' awareness, the store can show the app's risk score along with its recommendation value. A even better approach would be a customised application of the third-party store that can be installed in users' devices, to show all this information. This work will require additional effort and it is in our future plans.

References

[1] Smartphone OS Market Share, Q3 2014, http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Last access Jan. 2015.

[2] Worldwide tablet shipments miss targets as first quarter experiences single-digit growth, according to idc, http://www.idc.com/getdoc.jsp?containerId=prUS24833314. Last access Jan. 2015.

[3] INFOSECURITY-MAGAZINE (2014), 92% of Top 500 Android Apps Carry Security or Privacy Risk, http://www.infosecurity-magazine.com/news/92-of-top-500-android-apps-carry-security-or/.

[4] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S. and JIANG, X. (2012) Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12 (New York, NY, USA: ACM): 281–294. doi:10.1145/2307636.2307663, URL http://doi.acm.org/10.1145/2307636.2307663.

[5] FELT, A.P., CHIN, E., HANNA, S., SONG, D. and WAGNER, D. (2011) Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11 (New York, NY, USA: ACM): 627–638. doi:10.1145/2046707.2046779, URL http://doi.acm.org/10.1145/2046707.2046779.

[6] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y. *et al.* (2014) Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14 (New York, NY, USA: ACM): 259–269. doi:10.1145/2594291.2594299, URL http://doi.acm.org/10.1145/2594291.2594299.

[7] LU, L., LI, Z., WU, Z., LEE, W. and JIANG, G. (2012) Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12 (New York, NY, USA: ACM): 229–240. doi:10.1145/2382196.2382223, URL http://doi.acm.org/10.1145/2382196.2382223.

[8] GIBLER, C., CRUSSELL, J., ERICKSON, J. and CHEN, H. (2012) Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12 (Berlin, Heidelberg: Springer-Verlag): 291–307. doi:10.1007/978-3-642-30921-2_17, URL http://dx.doi.org/10.1007/978-3-642-30921-2_17.

[9] Yang, Z. and Yang, M. (2012) Leakminer: Detect information leakage on android with static taint analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*, WCSE '12 (Washington, DC, USA: IEEE Computer Society): 101–104. doi:10.1109/WCSE.2012.26, URL http://dx.doi.org/10.1109/WCSE.2012.26.

[10] Chin, E., Felt, A.P., Greenwood, K. and Wagner, D. (2011) Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11 (New York, NY, USA: ACM): 239–252. doi:10.1145/1999995.2000018, URL http://doi.acm.org/10.1145/1999995.2000018.

[11] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P. and Sheth, A.N. (2010) TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10 (Berkeley, CA, USA: USENIX Association): 1–6. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.

[12] Yan, L.K. and Yin, H. (2012) Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12 (Berkeley, CA, USA: USENIX Association): 29–29. URL http://dl.acm.org/citation.cfm?id=2362793.2362822.

[13] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S. *et al.* (2013) Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13 (New York, NY, USA: ACM): 611–622. doi:10.1145/2508859.2516689, URL http://doi.acm.org/10.1145/2508859.2516689.

[14] Zhou, Y., Wang, Z., Zhou, W. and Jiang, X. (2012) Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. *Proceedings of the 19th Annual Network and Distributed System Security Symposium* : 5–8.

[15] Chakradeo, S., Reaves, B., Traynor, P. and Enck, W. (2013) Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13 (New York, NY, USA: ACM): 13–24. doi:10.1145/2462096.2462100, URL http://doi.acm.org/10.1145/2462096.2462100.

[16] Frank, M., Dong, B., Felt, A. and Song, D. (2012) Mining permission request patterns from android and facebook applications. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*: 870–875. doi:10.1109/ICDM.2012.86.

[17] Ham, H.S. and Choi, M.J. (2013) Analysis of android malware detection performance using machine learning classifiers. In *ICT Convergence (ICTC), 2013 International Conference on*: 490–495. doi:10.1109/ICTC.2013.6675404.

[18] Rieck, K., Trinius, P., Willems, C. and Holz, T. (2011) Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.* **19**(4): 639–668. URL http://dl.acm.org/citation.cfm?id=2011216.2011217.

[19] AXMLPrinter, class in Java code, http://code.google.com/p/xml-apk-parser/source/browse/trunk/src/test/AXMLPrinter.java. Last access Oct. 2014.

[20] Xu, Z., Bai, K. and Zhu, S. (2012) TapLogger. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks - WISEC '12* (New York, New York, USA: ACM Press): 113. doi:10.1145/2185448.2185465, URL http://dx.doi.org/10.1145/2185448.2185465http://dl.acm.org/citation.cfm?doid=2185448.2185465.

[21] Mark Manning, Manitree: AndroidManifest.xml Auditor, https://github.com/antitree/manitree. Last access in Feb. 2014.

[22] Ben Gruver, Smali/baksmali, an assembler/disassembler for the dex format, http://code.google.com/p/smali/. Last access in Aug. 2014.

[23] Anthony, Desnos, Androguard, a python tool for reverse engineering, malware and goodware analysis of Android applications, http://code.google.com/p/androguard/. Last access in Aug. 2014.

[24] Google, Android permissions, http://developer.android.com/reference/android/Manifest.permission.html. Last access in Aug. 2014.

[25] Fahmida, Jailbreak exploit gingerly hits android with malware, http://www.techweekeurope.co.uk/workspace/jailbreak-exploit-hits-android-gingerbread-\with-malware-37712. Last access in Feb. 2015.

[26] Lantz, P. (2011) *An Android Application Sandbox for Dynamic Analysis*. Master thesis, Lund University. https://code.google.com/p/droidbox.

[27] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M. and Ioannidis, S. (2014) Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. *Proceedings of the Seventh European Workshop on System Security* doi:10.1145/2592791.2592796, URL http://doi.acm.org.acces.bibl.ulaval.ca/10.1145/2592791.2592796.

[28] Zadeh, L.A. (2008) Is there a need for fuzzy logic? In *Fuzzy Information Processing Society, 2008. NAFIPS 2008. Annual Meeting of the North American*: 1–3. doi:10.1109/NAFIPS.2008.4531354.

[29] Open source users' community (2012), An open-source api for the android market, http://code.google.com/p/android-market-api/. Last access in Aug. 2014.

[30] Zhou, Y. and Jiang, X. (2012) Dissecting Android Malware: Characterization and Evolution. In *2012 IEEE Symposium on Security and Privacy* (IEEE): 95–109. doi:10.1109/SP.2012.16, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6234407.

[31] Contagio, Contagio malware dump, http://contagiodump.blogspot.it/. Last access in Aug.

2014.

[32] Liccardi, I., Pato, J., Weitzner, D.J., Abelson, H. and De Roure, D. (2014) No technical understanding required: Helping users make informed choices about access to their personal data. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MOBIQUITOUS '14 (ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering)): 140–150. doi:10.4108/icst.mobiquitous.2014.258066, URL http://dx.doi.org/10.4108/icst.mobiquitous.2014.258066.

[33] Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C. *et al.* (2012) Using probabilistic generative models for ranking risks of Android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*, CCS '12 (New York, New York, USA: ACM Press): 241. doi:10.1145/2382196.2382224, URL http://doi.acm.org/10.1145/2382196.2382224http://dl.acm.org/citation.cfm?doid=2382196.2382224.

[34] Sarma, B.P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C. and Molloy, I. (2012) Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12 (New York, NY, USA: ACM): 13–22. doi:10.1145/2295136.2295141, URL http://doi.acm.org/10.1145/2295136.2295141.

[35] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J. and Le Traon, Y. (2013) Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13 (Berkeley, CA, USA: USENIX Association): 543–558. URL http://dl.acm.org/citation.cfm?id=2534766.2534813.

[36] Davi, L., Dmitrienko, A., Sadeghi, A.R. and Winandy, M. (2011) Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10 (Berlin, Heidelberg: Springer-Verlag): 346–360. URL http://dl.acm.org/citation.cfm?id=1949317.1949356.

[37] Grace, M.C., Zhou, W., Jiang, X. and Sadeghi, A.R. (2012) Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12 (New York, NY, USA: ACM): 101–112. doi:10.1145/2185448.2185464, URL http://doi.acm.org/10.1145/2185448.2185464.

[38] Zhou, W., Zhou, Y., Jiang, X. and Ning, P. (2012) Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12 (New York, NY, USA: ACM): 317–326. doi:10.1145/2133601.2133640, URL http://doi.acm.org/10.1145/2133601.2133640.

[39] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B. and Smith, M. (2012) Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12 (New York, NY, USA: ACM): 50–61. doi:10.1145/2382196.2382205, URL http://doi.acm.org/10.1145/2382196.2382205.

[40] Mallodroid will help identify vulnerable android-apps, http://malwarelist.net/2012/10/23/vulnerable-android-apps/. Last access in Feb. 2015.

[41] Roman Unuchek - Lookout, The most sophisticated android trojan, http://securelist.com/blog/research/35929/the-most-sophisticated-android-trojan/. Last access in Feb. 2015.

[42] Tim Strazzere, Android-unpacker, https://github.com/strazzere/android-unpacker. Last access in Feb. 2015.

Appendix

Table F.1. The mapping between behaviour patterns and their risk categories in *Behaviour* module.

Activity Name	Risk categories
APP_DATA	N/A
PERMISSIONS	DANGEROUS_RISK, MONEY_RISK, PRIVACY_RISK, SMS_RISK, ROOT_PRIV_RISK, EXPLOIT_RISK, INTERNET_RISK, PHONE_RISK
ADVM_PERMISSIONS	DANGEROUS_RISK, DYNAMIC_RISK, MONEY_RISK, PRIVACY_RISK, SMS_RISK, ROOT_PRIV_RISK, EXPLOIT_RISK, INTERNET_RISK, PHONE_RISK
UNUSED_PERMISSIONS	N/A
SIGNATURE_PERMISSIONS	SIGNATURE_SYSTEM_RISK or SIGNATURE_RISK
DYN_RCV	DANGEROUS_RISK, DYNAMIC_RISK
DYN_CLASS_LOAD	DANGEROUS_RISK, DYNAMIC_RISK, BINARY_RISK
RUNTIME_EXEC	DANGEROUS_RISK, EXPLOIT_RISK, ROOT_PRIV_RISK
IMPLICIT_HTTP	N/A
ENCRYPTION	DANGEROUS_RISK, ENCRYPTED_CODE_RISK
REFLECTION	DANGEROUS_RISK, BINARY_RISK, EXPLOIT_RISK
NATIVE	DANGEROUS_RISK, BINARY_RISK, EXPLOIT_RISK
PKG_INSTALL	DANGEROUS_RISK, BINARY_RISK
PKG_MONITOR	DANGEROUS_RISK
CALL_PHONE	DANGEROUS_RISK, MONEY_RISK
PROCESS_OUTGOING_CALLS	DANGEROUS_RISK, MONEY_RISK, PRIVACY_RISK
SEND_TXT_SMS	DANGEROUS_RISK, MONEY_RISK, MONEY_RISK
SEND_DATA_SMS	DANGEROUS_RISK, MONEY_RISK, MONEY_RISK
TEL_MANAGER	DANGEROUS_RISK, MONEY_RISK, MONEY_RISK
INCOMING_CALLS	DANGEROUS_RISK, PRIVACY_RISK
ACCESS_CALL_LOG	DANGEROUS_RISK, PRIVACY_RISK
RCV_DATA_SMS	DANGEROUS_RISK, PRIVACY_RISK
RCV_TXT_SMS	DANGEROUS_RISK, PRIVACY_RISK
READ_INBOX	DANGEROUS_RISK, PRIVACY_RISK
READ_SENT	DANGEROUS_RISK, PRIVACY_RISK
READ_OUTBOX	DANGEROUS_RISK, PRIVACY_RISK
READ_OTHERS	DANGEROUS_RISK, PRIVACY_RISK
HTTP_POST	DANGEROUS_RISK, PRIVACY_RISK, INTERNET_RISK, MONEY_RISK
HTTP_GET	DANGEROUS_RISK, PRIVACY_RISK, INTERNET_RISK, MONEY_RISK
HTTP	DANGEROUS_RISK, PRIVACY_RISK, INTERNET_RISK, MONEY_RISK
SOCKET	DANGEROUS_RISK, PRIVACY_RISK, INTERNET_RISK, MONEY_RISK
SRV_SOCKET	DANGEROUS_RISK, PRIVACY_RISK, INTERNET_RISK, MONEY_RISK
NET_MONITOR	DANGEROUS_RISK, PRIVACY_RISK, INTERNET_RISK
START_SRV	DANGEROUS_RISK, INTERNET_RISK
READ_CONTACTS	DANGEROUS_RISK, PRIVACY_RISK
ACCESS_FINE_LOCATION	PRIVACY_RISK
ACCESS_COARSE_LOCATION	PRIVACY_RISK
KILL	PHONE_RISK
RESTART	PHONE_RISK
CHANGED_SETTINGS	PHONE_RISK
READ_SETTINGS	PHONE_RISK
DEVICE_POLICY_MANAGER	DANGEROUS_RISK, PHONE_RISK, PRIVACY_RISK
WRITE_BOOKMARKS	DANGEROUS_RISK, PHONE_RISK
READ_BOOKMARKS	PRIVACY_RISK
WRITE_EXTERNAL_STORAGE	N/A
READ_SMS	DANGEROUS_RISK, PRIVACY_RISK, SMS_RISK

WRITE_SMS	DANGEROUS_RISK, PRIVACY_RISK, SMS_RISK
ADD_APN	DANGEROUS_RISK, PRIVACY_RISK, SMS_RISK, MONEY_RISK
READ_APN	N/A
PROTECTED_INTENTS	DANGEROUS_RISK, PHONE_RISK
MANIFEST	N/A
FILES	N/A
IP	N/A
URLS	N/A
CAMERA	PRIVACY_RISK
FACTORY_TEST	DANGEROUS_RISK, PHONE_RISK
NFC	PRIVACY_RISK
READ_INCOMING_SMS	PRIVACY_RISK, SMS_RISK
WRITE_CONTACTS	N/A
SMS_DEFAULT_APP	PHONE_RISK, SMS_RISK, PRIVACY_RISK
VPN_SERVICE	N/A
WIFIP2P	N/A
NFC_BEAM_SENDER	N/A
NFC_BEAM_RECEIVER	N/A

Table F.2. The mapping between detected activities and their risk categories in dynamic analysis module.

Activity Name	Risk categories
FILE_READ	PRIVACY_RISK, EXPLOIT_RISK, PHONE_RISK;
FILE_WRITE	EXPLOIT_RISK, PHONE_RISK;
CRYPTO_API	ENCRYPTED_CODE_RISK;
OPEN_CONN	INTERNET_RISK;
OUTGOING_TRAF	INTERNET_RISK, MONEY_RISK;
INCOMING_TRAF	INTERNET_RISK, MONEY_RISK;
DEX_CLASS_LOADER	DYNAMIC_RISK, BINARY_RISK;
BROADCAST_RECEIVER	DYNAMIC_RISK;
START_SERVICE	DYNAMIC_RISK, PHONE_RISK;
ENFORCED_PERMISSION	DYNAMIC_RISK;
BYPASSED_PERMISSION	DYNAMIC_RISK, PHONE_RISK;
SENT_SMS_NORMAL	SMS_RISK, MONEY_RISK;
PHONE_CALLS	MONEY_RISK;
FILE_LEAKAGE	PRIVACY_RISK, PHONE_RISK;
INTERNET_LEAKAGE	PRIVACY_RISK, INTERNET_RISK, MONEY_RISK;
SENT_SMS_LEAKAGE	PRIVACY_RISK, SMS_RISK, MONEY_RISK;
INFO_LEAKAGE_SMS	SMS_RISK, PRIVACY_RISK;
INFO_LEAKAGE_OTHERS	PHONE_RISK, PRIVACY_RISK.