How dangerous is your Android app? An evaluation 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, Torino, Italy
{shocked, tao.su}@polito.it
[2]Telecom Italia Lab, Via Reiss Romoli, 274, Torino, Italy
{madalina.baltatu, rosalia.dalessandro}@it.telecomitalia.it
[3]former student Politecnico di Torino, Italy
giovanni.pessiva@gmail.com

ABSTRACT

In the last decade, we have witnessed an unprecedented increase in the adoption of mobile devices. A substantial number of these devices run on the Android operating system. Android is an open-source operating system based on Linux, which provides a permission-based security model that demands each application to request explicit permissions (approved by the user) before it can be installed to run. However, end users cannot estimate application risk, so the user's decision is almost completely unrelated to the application risk level. Moreover, due to the platform openness and the plethora of available software, dangerous apps (even if not necessarily malware) are now also very common for Android devices.

In this paper we propose a new approach and a tool to evaluate the potential risk of Android application packages to help end user security awareness. The tool exploits both static and dynamic analysis techniques. It examines the correlations between app required permissions and the invoked APIs, as well as the contents in the package, and subsequently it uses a dynamic analysis module to confirm the suspicions proposed by static modules. The risk activities detected by analysis modules are then mapped into finer-grained risk categories and further evaluated using the fuzzy logic algorithm. Fuzzy logic aims to deal with uncertainty which arises from the nature of automatic analysis, as not all detected activities intend to cause harm. For the sake of both tech-uninterested and tech-savvy users, the results contain a simple numerical value showing the risk level plus a detailed report of detected activities and their mappings to the risk categories. Finally, we tested our software on a large set of real-world samples, demonstrating its efficiency and showing a reasonable capacity to identify and evaluate the potential risk of application packages, both the benign and the malicious ones.

MOBIQUITOUS 2014, December 02-05, London, Great Britain
Copyright © 2014 ICST 978-1-63190-039-6
DOI 10.4108/icst.mobiquitous.2014.257832

Keywords

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

1. INTRODUCTION

Android has become the most popular mobile operating system, it is installed on the vast majority of smartphones and a significant percentage of tablets. Every day, thousands of different apps are published through the official or third-party application repositories. As shown in practice, in this huge number of applications, a lot of them contain security and privacy risk [1], such as accessing the contacts, uploading current location and retrieving device information.

This kind of dangerous behaviour is common in both benign and malicious applications. Some are caused by developer's misjudgement, e.g. invoking suspicious ad-ware or recycled code. Some are caused to fulfil the requirements of application functionality. For example, instant message applications such as *Viber* usually require to access to contact list to find out who else is also using it. Moreover, as part of authentication mechanism, it uses mobile phone number as user identity. For this reason, in account activation process, *Viber* server will send a SMS to the phone number with an activation code. Then the app installed in mobile device accesses SMS and verifies the activation code to confirm that the user owns the phone number. This kind of operations on contact list and SMS is equally risky as in malware to security-sensitive users, even if in this case there is an acceptable reason; in scenarios like this one, the boundary between legitimate or malicious applications is blurred.

Android permission-based security model leaves to end users the management of accessing controls to device resources. But 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 a potential risk level of an Android app package (*apk*). The analysis output consists of a detailed app behaviour report and a simple numeric value as comprehensive risk estimation, that prior to *apk* installation can give a risk indication for both tech-savvy and common users.

A big challenge in building fully automatic analysis systems is how to evaluate the analysis results in order to present to end user 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 valid in Android app analysis environment. In all previous researches, static [2–8], dynamic [9–11] or hybrid [12] 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 types of threshold, then human inspection is required to categorise the rest samples. Although the filtering process will significantly reduce the cost, it is still inconvenient for a market-scale analysis. Further researches address this point, applying complex reasoning techniques (e.g. machine learning, data mining, ...) to allow analysers to make the final decision. We, on the contrary, try to use fuzzy logic algorithm to overcome this uncertainty limitation arises from 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 finer-grained risk categories;
- We evaluate application risk level using fuzzy logic algorithm, trying to overcome/mitigate the uncertainty limitation arose from the nature of automatic analysis;
- We implement a prototype system, 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 static and dynamic analysis modules, as well as 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 works on Android apps 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 is based on the Linux kernel, and inherits its security features, like the user-based permissions model used to control app execution. A unique Linux user identifier (UID) is assigned to every installed package; consequently, applications will run as that user in separate processes. In this way a kernel-level application sandbox is implemented.

Android apps are written in Java and run on a proprietary Virtual Machine called Dalvik (DVM). 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 resources (e.g., images and strings) are compiled with the command `aapt` into a single file.

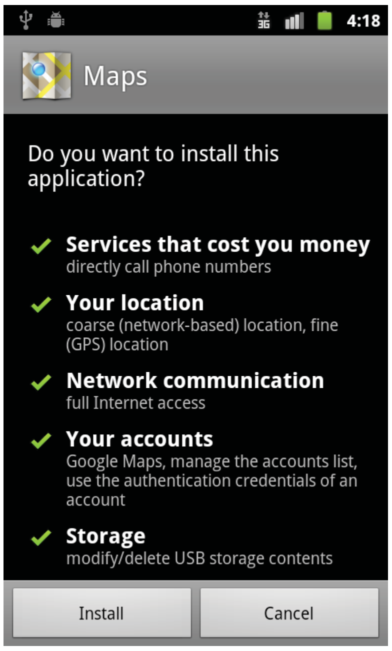
All the files are then packaged into an *apk* (Android Package) file, which is basically a *zip* compressed archive, using *apkbuilder*. This file is then signed with *jarsigner*, using a certificate generated and self-signed by the developer, which is checked only at install time by the system.

The basic content of an *apk* file is the following:

- a *META-INF* directory, that contains developer's certificate;
- a *res* directory, that contains raw resources;
- an optional *assets* directory, that contains application assets;
- *AndroidManifest.xml*, the manifest file in binary XML format;
- *classes.dex*, the source code of the application in Dalvik Executable format;
- *resources.arsc*, a file containing pre-compiled resources.

The manifest file contains the essential data, which the operating system needs in order to install and run the application. Of particular interest, the list of permissions required by the app and the used features (e.g., hardware sensors). The permissions limit what the app can do; each permission is identified by a unique label, for example `android.permission.SEND_SMS`, `android.permission.INTERNET`. If an application tries to use a feature whose permission is not granted, the system will terminate it.

At install time, the user is required to approve the permission list requested by the app, as shown in Fig. 1.



Do you want to install this application?

✓ **Services that cost you money**
directly call phone numbers

✓ **Your location**
coarse (network-based) location, fine (GPS) location

✓ **Network communication**
full Internet access

✓ **Your accounts**
Google Maps, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

Figure 1: Permissions requested by an application.

The permission model, while being intuitive for developers and users, has some flaws, and by itself is not enough to prevent malicious or risky operations. For example, *TapLogger* [13] 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 Android security mechanism is using dynamic code loading; the code can be prestored inside the *apk* or even downloaded from the Internet, and may contain malicious parts which are much more difficult to detect.

3. ANDROID APPLICATION EVALUATOR

In mobile environment, risky operations can easily cause privacy leaks or money losses for the device owner, due to vulnerabilities in software or users tendency to allow more permissions than needed (and developers inclination to ask for more than needed).

To better inform users decisions, our risk analyser approaches Android app risk estimation problem exploiting both static and dynamic detection techniques.

The more concise goal of our evaluator 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 users, who can subsequently choose how to manage the potential source of threat, e.g. carefully read the more detailed report our tool provides.

In our context, the word "risk" is used for alerting about a tangible danger (e.g., a privilege escalation is signaled when a specific shell command is present, money risk when SMS are sent or phone calls are made). Conservatively, it also flags a potentially dangerous situation (e.g., the presence of a generic embedded binary executable). Each situation is mapped to a specific risk category as indicated in the following subsections in different analysis modules.

3.1 Static Analysis

Static analysis is the process of analysing software applications without executing them. Usually the starting point is decompiling the applications 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; it is packaged inside the *apk* in binary XML format, and many tools exist to make it human readable (e.g. *AAPT*, the Android Asset Packaging Tool included in the Android SDK). Reading the manifest content, a number of tools can point out possible insecurities. For example, Manitree [14], among a number of others, searches for services shared with other apps in the device without an intent filter or a permission requirement, which would allow accessing from other apps; since malware often sets higher priority values to forerun other app requests, it also looks at intents and 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), 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 code. Different tools exist to dump Dalvik bytecode or to convert it to other low-level representations (e.g., Smali [15]); the result in general is easily understandable, unless obfuscation techniques have been used to harden the readability. On the basis of the analysis goal, the preferred human-readable representation could 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.

The static analysis is implemented through several modules, which leverage extensively on the *androguard* APIs [16], an effective set of tools written in Python for performing static analysis on Android applications. We extended it through two complementary modules, *Behaviour* and *FileScan*.

3.1.1 *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) critical APIs usage to finds out potential dangerous operations.

As first step, *Behaviour* scans the app's manifest, retrieving the permission list. Then it decompiles the app to obtain the source code. After that, the source code is analysed to find out what APIs are invoked and what operations the app attempts to execute. In this way, the tool can 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.

In order to detail the types of menace posed by dangerous operations, we enlarged the *androrisk* risk taxonomy and enable *Behaviour* module to map all Android permissions [17] to an augmented pool of risks. The source code which leads to dangerous activities are mapped to the following risk categories:

- Root privileges escalation
- Encrypted code
- Binary code
- Internet
- Dangerous API
- Dynamic code loading
- Exploit
- Phone abuse
- SMS activities
- Money risk
- Signature and system permission
- Privacy violation

The rationale behind this mapping is to enumerate and characterise the possible danger the user might face, and present this result in a user meaningful way. Some of the mappings are straightforward (e.g. the *root privileges escalation* activity is categorised into *Root privileges escalation* 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 not only based on the detected activities, but 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 on a more detailed basis.

3.1.2 *FileScan* module

FileScan, our second static analysis module, analyses every file stored inside the app package. It identifies the file type using the information contained in file header, such as magic number. In this way, *FileScan* module can detect dangerous files, like embedded app, infected files or shell scripts with suspicious commands.

Many malware apps attempt to conceal their purposes, and often alter file names to use some innocuous extension (e.g. *png*). For example the malware families of DroidDream and GingerMaster use this trivial technique. *FileScan*, through magic number analysis, can identify the dangerous files even 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.

Embedded apps are *apk* files stored inside the app package, that can be installed or loaded at runtime through dynamic code injection. Containing embedded application is dangerous since the secondary app may contain malicious code, and many static analysis systems are not able to detect and analyse them properly. The *elf binaries*, whether executables or shared libraries, can be used by the app for a direct access to the system APIs. *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.

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

- Hidden elf binary
- Hidden apk
- Hidden text
- Infected elf binary
- Infected dex code
- Input shell
- Shell install (Script with install commands)
- Shell privilege (Scripts with privilege escalation commands)
- Shell other (Scripts with other critical 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 SMS 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. 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 checks are needed. However, this shortcoming can be mitigated by combining the analysis result of dynamic analysis module, which outputs the phone numbers and URL addresses used during 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 still achieves what is currently the best possible result for an automated analysis of this type of menace.

3.2 Dynamic Analysis

Dynamic analysis is a run-time analysis of apps, 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 in a real environment or in an emulated one. Emulation is the cheaper solution, but it suffers some limitations. For example, the emulated environment does 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 path of emulation.

Our dynamic analysis module is developed on top of *Droidbox* [18], a well-known open-source dynamic analysis tool for Android applications. The module enriches *Droidbox* in a number of ways, from the input and output 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.

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

- Encrypted code
- Binary code
- Dynamic code loading
- Exploit risk
- Internet
- Money risk
- SMS activities
- Privacy violation
- Phone abuse

Our automatic dynamic analyser is very effective against risky apps which execute dangerous operations directly after they are installed and started by *adb*. If stealth techniques are used, for example a hidden trigger, our analyser, as most automatic systems, needs human interactions to bring the dangerous operations. Nevertheless, the risk scores it computes can offer realistic and reliable danger level estimations from a fully automatic analysis point of view.

Another drawback of dynamic analysis consists in its time-consuming nature. The emulator needs to start up for each *apk* with a clean Android virtual device image, and then it has to wait for the tested app to finish all its initial operations. 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

Fuzzy logic is widely used in decision making systems. As stated by Prof. Zadeh in [19], "*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, 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.

Consequently in our analyser we defined *risk score* as a final output of our system, which aims to give a quick indication to 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).

3.3.1 Fuzzy interpretation of risk states

The input of the fuzzy logic system are derived from the risk categories and their corresponding violation frequency 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. For each risk category, we defined four separated states 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 end users while it can be easily interpreted using 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.

For instance, if the violation occurrence is 7, then `BINARY_RISK` is 75% in `LOW` risk state and 25% in `AVERAGE` state. If scoring system need to be tuned that it gives more weight to `BINARY_RISK`, the adjectives for each risk state can

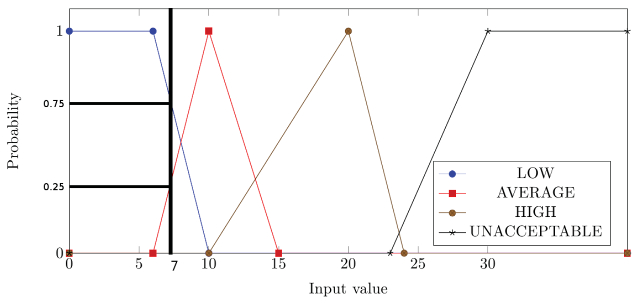


Figure 2: Adjectives defined for `BINARY_RISK`.

be reduced hence `BINARY_RISK` reaches `UNACCEPTABLE` state with less violation occurrence.

Selecting the boundaries for these adjectives have been challenging. The expected outcome is the realistic risk level of analysed samples; 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 was meaningful for end users.

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

3.3.2 Computing fuzzy risk level

To combine the states of all risk categories in scoring system, 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 30 rules aggregated in these three analysis modules, and some of them are refined by up to 5 sub-rules. 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 violation occurrence is 7, and this is the only rule in the scoring system, the risk score will be:

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

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 would be

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

The last step to compute the risk score is *defuzzification*, which can be performed in several different ways. 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 $\mathrm{u}_{\mathrm{A}}(\chi)$ indicates the `truth level` for all the attributes, and x is the adjectives for each attributes.

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.

For the rules with `FALSE` condition, they will be ignored. 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 app may have more violation occurrence in certain risk category. So 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.

4. EXPERIMENTAL RESULTS

To perform an extensive testing of our system, we developed an additional software module, the *AppsDownloader*, which is based on the unofficial open source project named Android Market API [20]; it can automatically retrieve free apps from any Android repositories and also from the local file system.

Exploiting the AppsDownloader and the workflow described in Fig. 3, we tested our analyser against a set of 41000 free goodware applications from *Google Play* (this set will be referred to as *market*); and a set of 1488 known malware samples from 90 distinct families, from the *Android Genome Project* [21] and *ContagioMiniDump* [22] (this set will be referred to as *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 [3], that 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 exceed this threshold will be considered risky in the case of permission abuse and calling

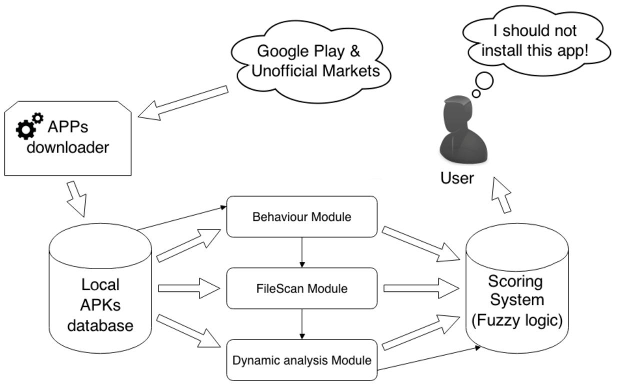


Figure 3: Danger level evaluator testing architecture.

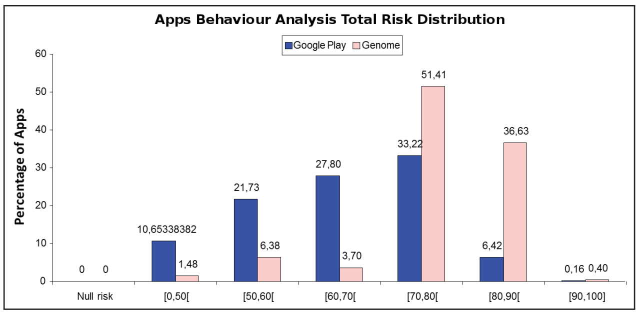


Figure 4: *Behaviour* module results: app risk scores distribution, market vs. malware.

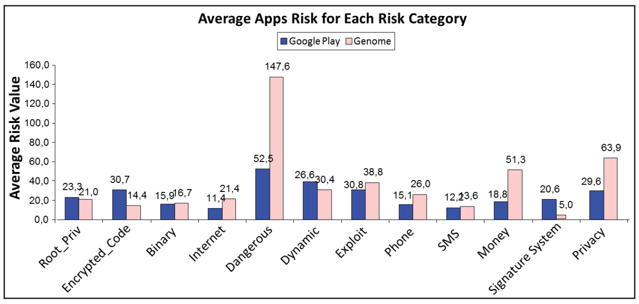


Figure 5: *Behaviour* module results: average app risk value per category, market vs. malware.

potentially dangerous APIs. From the risk score distribution, we can also see that scores of free applications are concentrated in the interval from 60 to 80, while 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.

Then, *FileScan* module tested both sets. The result is shown in Fig. 6, 99% of the *market apks* has a null risk score, while 40% of the *malware apks* has a risk greater than 80. The distribution of applications on *FileScan* risks categories is shown in Fig. 7, we can see that the `HiddenElf`, `ShellPrivilege` and `HiddenText` are the most violated risk

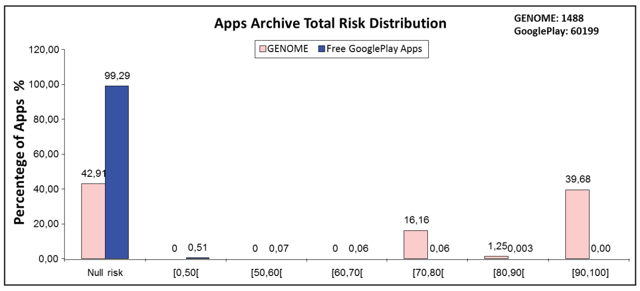


Figure 6: *FileScan* module results: app risk scores distribution, market vs. malware.

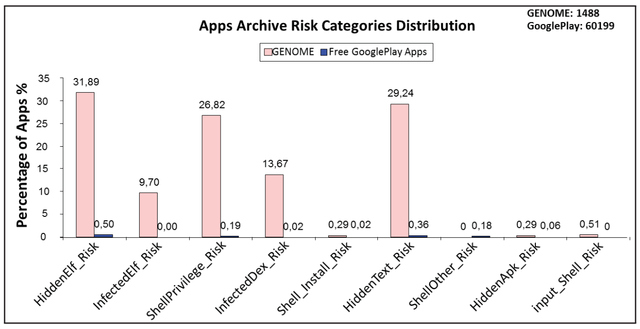


Figure 7: *FileScan* module results: app percentage per violating risk category, market vs. malware.

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.

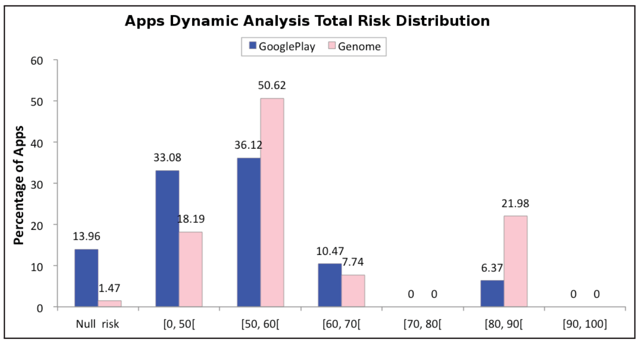


Figure 8: *Dynamic analysis* module result: app risk scores distribution, market vs. malware.

Dynamic analysis works as a confirmation mechanism, to prove the dangerousness of suspicious apps. The results for dynamic analysis alone are shown in Fig. 8 and Fig. 9. 50% of market *apks* obtained a risk score greater than 50, while 80% of malware *apks* exceed the same threshold. Shown in Fig. 9, almost all known malware has violated 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. And only known malware exploits the activities related to `BINARY` (`BaseBridge` samples) and `SMS` (`HippoSMS` and `FakePlayer` samples) risk. Moreover, the phone numbers, URL

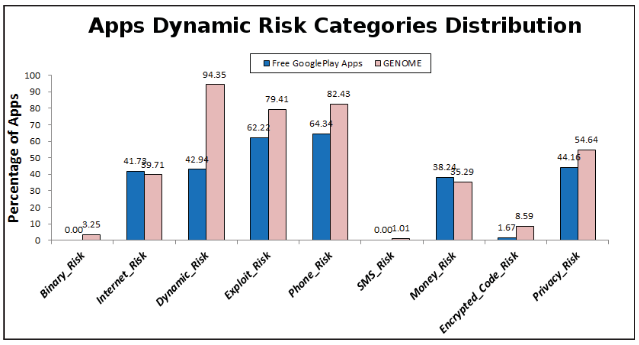


Figure 9: *Dynamic analysis* module result: app violation percentage per risk category, market vs. malware.

addresses and the activities detected during execution are valuable, since they indicate real behaviour of the analysed samples after installation and initiation.

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

Besides risk scores, the analysis system exposes and confirms 288 suspect URL addresses and 5 phone numbers identified in the tested applications. We found out 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 from malware. Regarding the misusage of SMS and phone calls, only certain known malware tries these unauthorised communications, since they are pretty easy to be detected by users.

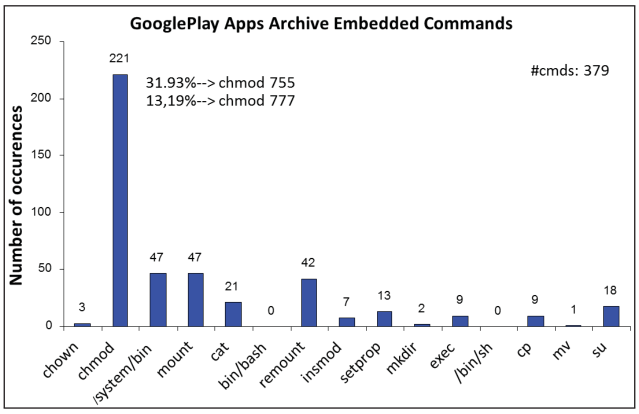


Figure 10: embedded commands in market apps.

We also investigated the most common shell commands encountered in the apps from *market* and *malware* datasets. The results are presented in Fig. 10 and Fig. 11. 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.

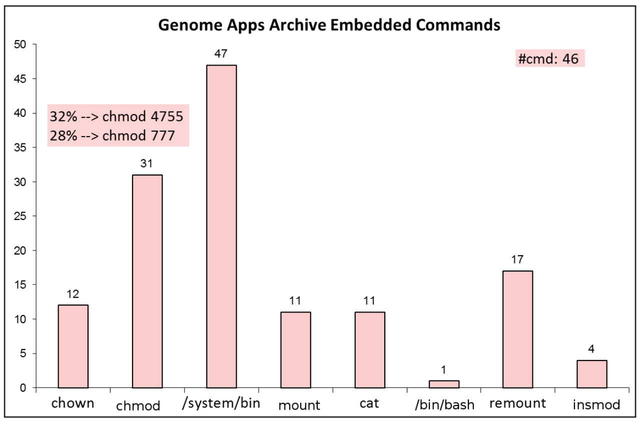


Figure 11: embedded commands in malware apps.

A significant result of our tests is that the *market apks* could not be as innocuous as users are inclined to think. This may be an effect of a poor programming or presence of potentially unwanted code (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* in attractive and powerful targets for Trojans, which can exploit these over-permissioned *apks* to unrestricted act on end user devices.

5. RELATED WORK

Our analyser aims to analyse a market-scale number of applications without any human interaction, and try to overcome the uncertainty arises from the nature of automatic analysis. Its goal is to highlight the risk level 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` or developers' tendencies to require more than needed, which easily create risks for users. Thus, many methodologies check the requested permission in search of risks of misbehaviour. In [23], 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 result. Another tool in this category is *Stowaway* [3]; 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 [24]. The authors evaluate app risks on the basis of how rarely permissions are required for apps in a specific category, like navigating or games. Since Android's permission model fails to fully control application behaviour. Thus, analysing solely the permissions requested can only be a start point, but it is incomplete when evaluating the application risks.

Except permission-based, other static analysis approaches are also proposed. *Taint analysis* addresses the problem of analysing Android apps based on their data flows. In [4], 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 of data leakage. *CHEX* [5], *AndroidLeaks* [6], *LeakMiner* [7] all use the same static taint analysis approach to analyse data leakage caused by Android apps. Intercomponent communication (i.e. ICC) is also a studying point to analyse Android apps. In [25], 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 risky operations such as transitive privilege usage [26]. Similarly, *ComDroid* [8] also attempts to identify security risk in Android apps by analysing inter-application communications. However, as in the permission-based approach, the analysis results using previous methods can only partially cover 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 efforts, but few of them fully understand the risk implication or fully control the behaviour. AdRisk [27] 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 [2], 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 [12] 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 in 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 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 fuzzy logic scoring algorithm.

Dynamic analysis techniques follow another path. TaintDroid [9] 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* [10] 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* [11], on the contrary, reconstructs app's behaviour with permission use analysis. Dynamic analysis requires a significant amount of time, usually not 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* [28] uses Multiple Correspondence Analysis (MCA) technique to measure the correlation between declared indicators of functionalities to be presented in app's package. It needs a large dataset (including both benign and malware samples) as training data, then applies the correlations to the analysed samples. Similarly, in [29], the authors apply pattern mining technique to permissions 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 fuzzy logic algorithm is simpler and straightforward. Also the cost is fair; each computation takes only around two 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 (be it malware or not), to inform 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.

The system can execute static and dynamic analysis separately or in cascade. This allows for flexibility in a number of scenarios. If time is constrained, 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 for 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 period strategies planning to contrast malware diffusion. On the other, this highlights excessive amounts of required permissions from app developers, and is a flag to demand more security-aware application development guidelines.

Future developments will include improvements in 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.

7. REFERENCES

[1] infosecurity-magazine: 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/

[2] Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: 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 (2012) 281–294

[3] Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS '11, New York, NY, USA, ACM (2011) 627–638

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: 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 (2014) 259–269

[5] Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: 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 (2012) 229–240

[6] Gibler, C., Crussell, J., Erickson, J., Chen, H.: 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 (2012) 291–307

[7] Yang, Z., Yang, M.: 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 (2012) 101–104

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

[9] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: 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 (2010) 1–6

[10] Yan, L.K., Yin, H.: 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 (2012) 29–29

[11] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., Zang, B.: Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security. CCS '13, New York, NY, USA, ACM (2013) 611–622

[12] Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: 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 (2012) 5–8

[13] Xu, Z., Bai, K., Zhu, S.: 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 (2012) 113

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

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

[16] Anthony, D., Androguard: 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.

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

[18] Lantz, P.: An Android Application Sandbox for Dynamic Analysis. Master thesis, Lund University (November 2011)

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

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

[21] Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. In: 2012 IEEE Symposium on Security and Privacy, IEEE (May 2012) 95–109

[22] Contagio: Contagio malware dump. http://contagiodump.blogspot.it/ last access in Aug. 2014.

[23] Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., Molloy, I.: 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 (2012) 241

[24] Sarma, B.P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C., Molloy, I.: 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 (2012) 13–22

[25] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: 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 (2013) 543–558

[26] Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Proceedings of the 13th International Conference on Information Security. ISC'10, Berlin, Heidelberg, Springer-Verlag (2011) 346–360

[27] Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.R.: 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 (2012) 101–112

[28] Chakradeo, S., Reaves, B., Traynor, P., Enck, W.: 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 (2013) 13–24

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