Risk Analysis of Android Applications: A User-Centric Solution

Gianluca Dini[b], Fabio Martinelli[a], Ilaria Matteucci[a], Marinella Petrocchi[a], Andrea Saracino[a], Daniele Sgandurra[c]

[a]*Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy*
[b]*Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Pisa, Italy*
[c]*Department of Computing, Imperial College London, United Kingdom*

Abstract

Android applications (apps) pose many risks to their users, e.g., by including code that may threaten user privacy or system integrity. Most of the current security countermeasures for detecting dangerous apps show some weaknesses, mainly related to users' understanding and acceptance. Hence, users would benefit from an effective but simple technique that indicates whether an app is safe or risky to be installed. In this paper, we present MAETROID (Multi-criteria App Evaluator of TRust for AndrOID), a framework to evaluate the trustworthiness of Android apps, i.e., the amount of risk they pose to users, e.g., in terms of confidentiality and integrity. MAETROID performs a multi-criteria analysis of an app at deploy-time and returns a single easy-to-understand evaluation of the app's risk level (i.e., Trusted, Medium Risk, and High Risk), aimed at driving the user decision on whether or not installing a new app. The criteria include the set of requested permissions and a set of metadata retrieved from the marketplace, denoting the app quality and popularity. We have tested MAETROID on a set of 11,000 apps both coming from Google Play and from a database of known malicious apps. The results show a good accuracy in both identifying the malicious apps and in terms of false positive rate.

Email addresses: `gianluca.dini@iet.unipi.it` (Gianluca Dini), `fabio.martinelli@iit.cnr.it` (Fabio Martinelli), `ilaria.matteucci@iit.cnr.it` (Ilaria Matteucci), `marinella.petrocchi@iit.cnr.it` (Marinella Petrocchi), `andrea.saracino@iit.cnr.it` (Andrea Saracino), `d.sgandurra@imperial.ac.uk` (Daniele Sgandurra)

Keywords: Android Security, User Experience and Expectations, Malware, Usability, Risk Analysis, User-Centric Devices.

1. Introduction

Smartphones and tablets allow users to access desired services everywhere, at any time. According to the recent market analysis in [1], in several countries more than 90% of registered mobile devices are smartphones or tablets, leading to more than 2 billions active subscribers. Noticeably, more than 80% of these devices are based on the Android operating system (OS), the most popular OS for smartphones and tablets [2]. Such a dominance of use makes Android also the almost exclusive target for mobile threats identified in the last years [3]. Currently, 99% of the Android security attacks are brought through infected mobile apps [3]. In particular, in 2014 Android accounted for 97% of all mobile malware [4]. Moreover, the number of new malware is alarming: on average, more than 160,000 new specimens are reported everyday [5]. Recently, work in [6] pointed out that a quarter of all Google Play free apps are clones, i.e., repackaged apps of popular ones, such as WhatsApp and Angry Birds.

Currently, apps for mobile devices are distributed through online marketplaces, such as *Google Play* or *App Store*. These marketplaces act as an hub where app developers publish their own products, which can be bought or downloaded for free by users. While official markets may charge users for these apps, several unofficial marketplaces distribute their own apps free of charge. When downloading apps from unofficial markets, trust is at risk, since there is no centralized control, as it happens with official markets, and it may happen that untrusted developers distribute malicious apps.

MAETROID statically evaluates the trustworthiness of an app at deploy time, before installation, starting from a set of criteria that are combined by means of a multi-criteria decision method called Analytic Hierarchy Process (AHP). As a result of the evaluation, MAETROID labels the app in a user-friendly way as either *trusted*, *medium risk* or *high risk*. MAETROID considers both objective and subjective criteria. The objective criteria consist in i) a global threat score deriving set of permissions required by the app, ii) the number of downloads of the application, and, iii) the market the app has been downloaded from. The subjective criteria consist in i) the app rating and ii) developer reputation.

The contributions of this paper are:

- A static evaluation method that exploits the app metadata and does not require code analysis. Our approach has shown to be effective without incurring into the complications and limitations of code analysis.

- A set of evaluation criteria for app evaluation. While other criteria might be conceived, we experimentally demonstrate that the set we propose is both reasonable and effective.

- MAETROID has been tested against a set of more than 11,000 apps, coming either from Google Play, unofficial markets and two important mobile malware database, namely Genome [7] and Contagio[1];

- The implementation of MAETROID for Android devices, downloadable from: `http://icaremobile.iit.cnr.it/`.

With respect to our previous work [8], the current work's novel contributions are a detailed investigation and formalization of the global threat score deriving from the permissions list required by the application, the detailing of the implementation of MAETROID as an Android app, and the results of the classification extended to a set of 11,000 apps.

Structure of the Paper. Section 2 recalls some notions on Android security mechanisms. Sect. 3 describes MAETROID by discussing in detail the criteria used for assessing the trustworthiness of an app. The current implementation of MAETROID for Android devices is presented in Section 4, which also presents the results of the analysis on the testbed apps. In Section 5, advantages and limitations of the MAETROID are reported, also in comparison with alternative solutions. In Sect. 6, we discuss related work on the security of mobile devices. Finally, Sect. 7 draws some conclusions and proposes some future research directions.

2. Android Permission System

To reduce the likelihood that a user installs a dangerous app, Android implements an access control mechanism called *Permission System.* The Permission System forces app developers to declare the security critical resources that the app can access and the security critical operations that the app can

[1]http://contagiominidump.blogspot.it/

perform. At run-time, an Android component called *Permission Checker* monitors the access requests to security critical resources and operations. If an access request is issued by an app without authorization declared in the Permission System, the permission checker denies the access.

During the period 2009-2015, starting with the original set of 90 permissions, a further set of 55 permissions has been added, mainly due to new device resources and apps functionalities. Currently, Android defines 136 permissions[2], where each permission is related to either a specific device resource or a critical operation. Permissions required by an app are declared by the developer in the `AndroidManifest.xml` file (*manifest*, for short), which is included in the app package (`apk`), bound to the app code by means of digital signature. Android classifies permissions in four classes: *normal*, *dangerous*, *signature*, and *signature-or-system*. For the scope of this paper, we only focus on normal and dangerous permissions, since signature and signature-or-system permissions cannot be required by any not-Google app. In fact, only apps signed with the Google private key (`signature`), thus developed by Google, or with specific authorizations released from Google (`signature-or-system`), can declare those permissions. The rationale behind signature and signature-or-system permission is that Google is directly interested in providing only genuine apps. This permission classification is used to choose which permissions are shown to the user at deploy-time. In fact, all dangerous permissions are automatically shown to the user, whereas the normal ones are listed in a separate sub-list (the "Other Permissions" list). Once a permission has been granted, the app can access the corresponding protected resource (or perform some corresponding critical operations) without asking for further authorizations. On the other hand, if a permission has not been declared in the app manifest, the application will never be allowed to access the resource or operation protected by that permission.

Figure 1 reports a time-line of the Android evolution, in terms of permission and vulnerabilities, during the time-span 2009-2015, by highlighting the major changes concerning permissions and security. In particular, until 2011 malware for Android systems were only confined to research proof-of-concepts. In 2011, Android-specific malware started to spread, by exploiting

[2]`http://developer.android.com/reference/android/Manifest.permission.html`

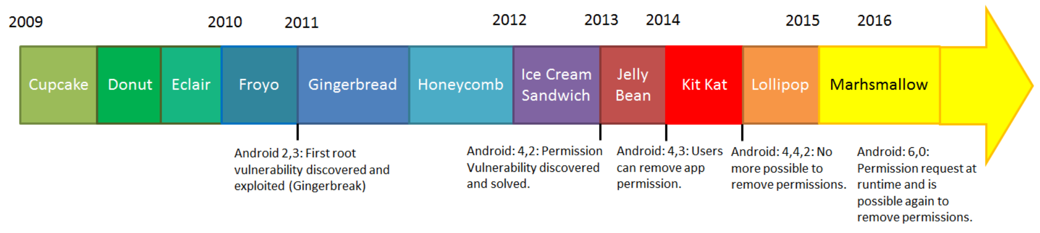

2009
2010
2011
2012
2013
2014
2015
2016
Cupcake
Donut
Eclair
Froyo
Gingerbread
Honeycomb
Ice Cream Sandwich
Jelly Bean
Kit Kat
Lollipop
Marhsmallow
Android 2,3: First root vulnerability discovered and exploited (Gingerbreak)
Android: 4,2: Permission Vulnerability discovered and solved.
Android: 4,3: Users can remove app permission.
Android: 4,4,2: No more possible to remove permissions.
Android: 6,0: Permission request at runtime and is possible again to remove permissions.

Figure 1: Time-line of Major Changes in the Android Permission System and Vulnerabilities

a vulnerability of the `Gingerbread` system, named `Gingerbreak`. This vulnerability allowed malicious apps to get root privileges, enabling them to potentially take the control of the device. This issue has been fixed with the introduction of Android `Ice Cream Sandwich` in 2012. However, in 2013, another important permission vulnerability has been discovered [9]. This vulnerability allows an attacker to modify the app permissions without modifying the app signature, mining thus the main pillar of Android security. An attempt to solve this issue has been proposed in the release of `Jelly Bean` (2013), which also includes for the first time the possibility to dynamically and selectively revoke, or grant, permissions to apps, through the `AppOps` permission manager.

Several criticisms have been raised against the Android permission system. Firstly, the system is considered too coarse-grained [10], since the user can only choose whether to accept all of the permissions declared by an app or to refuse to install the app. Secondly, users are generally unease at determining if an app can be trusted to be secure or not. In fact, the list of the required permissions is not very user-friendly, and it is quite difficult to fully understand the risk posed by such permissions. Since the number of requested permissions is rather large, and since some of them are quite difficult to understand, even for expert users, several users simply ignore them when installing a new app, leading to malicious apps being installed [11]. The aforementioned `AppOps` feature included in Jelly Bean, which allows users to revoke selected permissions to an already installed app, still presents issues related to the coarse granularity of the permissions system. In fact, any time the app tries to perform an operation for which the permission has been revoked, the operation is denied by the permission checker. Thus, since the error coming from the denied operation is not handled, the app is likely to terminate with error (it crashes).

It is worth noting that AppOps has been removed starting from the third Kit-Kat release (Android 4.4.2). Google claimed to remove it because meant for experimental purpose and misuse can break apps[3]. Indeed, as discussed, removing needed permissions will likely cause an app to crash. Thus, the decision to remove a permission should not be left to users which often lacks the required expertise for taking such a decision. With Android Lollipop (version 5) new permissions have been added to handle multi-users profiles and to give a stronger control on social network information sharing. The new version of Android (Marshmallow, version 6) brings a further improvement, allowing developers to require permissions not at deploy-time, but when the permission is effectively used, leaving to the user the decision to grant or not the authorization. However, some issues have already raised for this paradigm, since the procedure of granting permissions one by one may result cumbersome for apps with several permissions (e.g., the famous messaging app `Whatsapp` requires more than 20 permissions). Moreover, postponing the decision does not solve the problem of difficult permission interpretation. Another issue is that often developers declare (by mistake or for convenience) more permissions than those actually necessary, leading to the so called *Permission Overdeclaration* [12]. This happens because some permissions have similar names and their description is not self-explicative for some developers. It is quite intuitive that users, seeing a very long permissions list, are less encouraged to read and understand them.

These current limits of the Android permission system motivate the design of a user-centric advisory system, which analyzes the application security threat, providing to the user advises on whether it is safe or not to install a specific app.

3. MAETROID Design

To assess the apps' trustworthiness, MAETROID builds on a specific and customized instantiation of a well-known multi-criteria decision making process. We adopt the Analytic Hierarchy Process (AHP) [13]: given a decision problem, where several different *alternatives* can be chosen to reach a *goal*, AHP returns the *most relevant* alternative with respect to a set of *criteria*. The decision problem is structured as a hierarchy, by linking goals

[3]http://www.cnet.com/news/why-android-wont-be-getting-app-ops-anytime-soon/

Table 1: Fundamental Scale for AHP

Intensity	**Definition**	**Explanation**
1	Equal	Two alternatives contribute equally to the criterion
3	Moderate	One alternative is slightly more relevant than another
5	Strong	One alternative is strongly more relevant than another
7	Very strong	One alternative is very strongly more relevant than another
9	Extreme	One alternative is extremely more relevant than another

and alternatives through the criteria. AHP subdivides a complex problem into a set of sub-problems. Then, the most relevant alternative, i.e., the best solution for the decision problem, is computed by properly merging the local solutions.

Pairwise Comparison Matrices. Local solutions are computed by means of *pairwise comparison matrices.* Given a criterion, a pairwise comparison matrix describes how much an alternative is more relevant with respect to another one, in a pairwise fashion. It is a square matrix $n \times n$ (where n is the number of alternatives), which has positive entries and it is reciprocal, i.e., for each element a_{ij}, $a_{ij} = \frac{1}{a_{ji}}$, where $a_{ij} \in \{1, ..., 9\}$. For instance if we consider that the criterion i is five times more relevant than the criterion j, $a_{i,j}$ is equal to 5, and consequently $a_{j,i}$ is equal to its reciprocal $\frac{1}{5}$. Similarly, given the goal, a comparison matrix describes how much a criterion is more relevant with respect to another one. Again, it is a square matrix $k \times k$ (where k is the number of criteria), with positive entries and reciprocal. Table 1 shows the standard scale adopted in AHP to weigh alternatives with respect to the criteria. The same scale holds to weigh criteria with respect to the goal.

Computing Local Priorities. Local priorities express the relevance of 1) the alternatives for a specific criterion, and 2) the criteria for the specific goal. Given a comparison matrix, local priorities are computed as the normalized eigenvector associated with the maximum eigenvalue of the matrix [14]. Thus, for each criterion c_j, AHP extracts from the criteria comparison matrix a vector p_{c_j} of size n expressing the relevance, in percentage, of each alternative for that criterion. Similarly, for the goal comparison matrix, a vector p_g of size k is computed, expressing the relevance, in percentage, of each criterion for the goal.

Computing Global Priorities. Global priorities are computed through a weighted sum of all the local priorities calculated over the whole hierarchy (from alternatives to goal):

$$P_g^{a_i} = \sum_{j=1}^{k} p_g^{c_j} \cdot p_{c_j}^{a_i} \tag{1}$$

In the formula, $P_g^{a_i}$ is the global priority of the alternative a_i, $p_g^{c_j}$ is the local priority of criterion c_j with respect to the goal and $p_{c_j}^{a_i}$ is the local priority of alternative a_i with respect to criterion c_j.

Instantiation of goal, alternatives, and criteria in MAETROID. The goal of MAETROID consists in assigning one of the following alternative labels to the app under investigation:

Trusted: The app works correctly and does not hide malicious functionalities. Intuitively, a trusted app is characterized by a low global threat score, i.e., it is considered not being able to harm the system, due to the low threat of the required permissions. Moreover, a trusted app generally comes from the official market, it has been downloaded by thousands of users, it has very good reviews and/or has been developed by a developer with an outstanding reputation (i.e., a so called "top" developer). All the aforementioned features shape an app, as both secure and appreciated by the users. Therefore, the user could reasonably install the app without meeting with risks.

Medium Risk: The app may not work correctly and may include undesired, or malicious, functionalities. An app is considered to feature medium risks, in terms of the device security, when, even if it shows an acceptable (low) global threat score, it has received poor reviews, and/or it has been downloaded by too few users (less than 100) to infer that the app does not hide threats. Generally, this label should be assigned to low quality apps, published on official or unofficial markets, by non-skilled developers. Another reason for the assignment could be that the app is likely to be unwanted from the user, such as an advertising-supported software (Adware).

High Risk: The app likely includes malicious code. This label is given to apps that ask for several dangerous permissions, representing a potential threat to the device and the user. In fact, the largest majority of malware (95% [3]) asks for many suspicious permissions related to text messages, which should reasonably not be asked by apps unrelated to instant messaging. As highlighted in the following, apps that genuinely ask for several dangerous permissions are correctly evaluated as "Trusted" by MAETROID, mainly due to the positive user reviews combined with a conspicuous number of downloads (more than 100,000).

For MAETROID, we have defined five criteria, namely (i) a global threat score, obtained according to the declared permissions of the application, (ii) the marketplace, (iii) the developer's reputation, (iv) the user's rating, and (v) the number of downloads. In particular, the global threat score is calculated in terms of the amount and type of the declared permissions, on the basis of the resource the application may access and the operations it may perform. In the following, we detail all these criteria.

3.1. Threat score

We describe the scheme we have devised to compute the app *global threat score*, which is based on a weighted and normalized sum of single threat scores computed by manually evaluating all the permissions defined by Android. The single scores have been defined by referring to the resources and operations that can be accessed when each permission is granted. In detail, for each permission, we have evaluated the level of threat with respect to three security parameters: *privacy* (considering threats to user's data confidentiality), *system* (considering threats to the device integrity), and the *financial* threats (against the user's mobile credit). Table 2 shows the threats levels associable to such parameters.

The values in the table have been adapted from the standard for privacy risk management by CNIL (Commission Nationale de l'Informatique et des Libertés) [15], which describes the potential impact on privacy due to the misuse of digital resources. Here, we have extended the CNIL indexes also to system and financial threats. Consequently, it is possible to assign to each permission a score, which is representative of the threat it brings to the three security parameters. Finally, the global threat score is computed

Table 2: Threat Levels

Given Score	Meaning
0	No Threat
0.2	Negligible
0.4	Limited
0.6	Significant
0.8	Relevant
1	Maximum

by combining, weighting and normalizing, the three single scores of all the permissions declared by an application.

To explain the rationale behind our analysis of permissions, let us consider the permission `CALL_PHONE` as an example. We have assigned a score of 0.6 to its privacy threat, since this permission allows an app to operate phone calls without the user awareness. For example, this permission enables an app to start a phone call, without user consent to remotely record, on the called number, the user activities and nearby sounds to infer her behaviour. We have further assigned a score of 0.2 to the system threat, since the phone drains the battery faster during a call, but it is unlikely that an attacker can exploit phone calls to attack the device integrity by reducing the battery lifetime. Finally, we have assigned a score of 1 to the financial threat, since an app with this permission can call any phone number, including premium-rate numbers. Hence, a malicious app can cause a consistent financial damage to the user, e.g., by issuing calls to premium numbers which are likely to pass unnoticed (at least until the user credit ends or the user receives the bill).

Table A.7 in Appendix A gives an excerpt of the threat scores we have assigned to all the Android permissions (we refer the reader to [16] for a full list). In Table A.7, acronyms **PT**, **ST**, and **FT** are an abbreviation of *privacy*, *system*, and *financial* threat, respectively. The threat values have been given according to the documentation associated with all the permissions, which gives details on how they can be exploited by an app. The rationale behind the assignment of threat values to permissions is detailed in the following. Following the CNIL standard [15], a high level description of each threat level is also presented for all the three parameters.

3.1.1. Privacy threat score

Personal data, i.e., any information referring to an identified, or identifiable, individual (the so called *data subject*)[4], are governed by principles to guarantee their privacy [17]. As an example, the *Use Limitation Principle* states that "*Personal data should not be disclosed, made available or otherwise used for purposes other than those specified at the time of data collection except: i) with the consent of the data subject, or ii) by law.*". Following the principle, we define the privacy threat values for each permission as follows:

- **No Threat**: The permission does not affect users' data governed by privacy principles.
- **Negligible**: The permission affects data that either are not governed by privacy principles or whose violation let the data subject encounter really few inconveniences. Some examples are the active applications and connectivity interfaces (Wi-Fi) of the device.
- **Limited**: The permission affects data whose violation would lead to limited inconveniences, since, e.g., the disclosed information has poor accuracy and it does not strongly affect the user's privacy. An example is the `ACCESS_COARSE_LOCATION` permission, which gives access to the user location with a city-level granularity.
- **Significant**: The permission affects data whose violation leads to a significant inconvenience, with serious potential consequences. This value is assigned to permissions affecting data with serious privacy implications that however cannot be disclosed without a synergy with other permissions. An example is the `WRITE_SOCIAL_STREAM` permission, which grants the possibility to modify social network streams, but it does not allow to read them, thus limiting the privacy leakage issues.
- **Relevant**: The permission regulates the access to a resource or operation with serious privacy implications, which may definitively affect the privacy of the user. An example is the `ACCESS_FINE_LOCATION` permission, which grants full access to the user precise location.

[4]http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML

- **Maximum**: The permission regulates the access to a resource or operation seriously affecting both the privacy of the device's owner and of other users connected to that owner. The disclosure can cause serious damage to these users and the consequences are likely to be not recoverable. An example is the `READ_SMS` permission, which gives full access to all SMS messages received by the device's owner. An application with this permission can read both the SMS text and the number of the sender.

Examples of actions that may compromise or shatter the privacy principles are accessing the user's contacts, files, Internet bookmarks and chronology, as well as IMEI and IMSI codes. Permissions referring to these actions have received a high value for the privacy index. In particular, based on the principles highlighted in [17], we have given the highest privacy threat value to the following permissions:

- `android.permission.READ_CONTACTS`: it allows an app to access the contact list on the device. This may create a serious privacy leakage, since the app can send the data in the contact list to an attacker through Internet or text message.

- `android.permission.READ_PROFILE`: it allows an app to access data of the user account, which may contain private information, like birth-date, location, occupation. Apps with this permission can read, store, and eventually send them outside the user device.

- `android.permission.READ_SMS`: it allows an app to access SMS messages. Apps are able to read both the text and the sender number of all the private SMS messages stored in the device.

- `android.permission.RECEIVE_SMS`: it allows an app to control and handle the event of incoming SMS messages. This permission leads to privacy risk similar to that of the `READ_SMS` permission, even if, in this case, the app can only intercept incoming messages and it is not able to access the messages history. Remarkably, the app with such a permission granted may even decide not to show notifications of the received message to the user.

- `android.permission.RECEIVE_MMS`: it allows an app to control and handle the event of incoming multimedia messages. Same considerations of the above permission hold, applied to multimedia messages.

3.1.2. System threat score

A high value to system threat is assigned to those permissions accessing system components and potentially causing integrity issues to the OS, to files, and to the physical device. The interpretation of the indexes that we have given for the system threat is:

- **No Threat**: The permission does not regulate the access to any resource or operation affecting the device integrity.
- **Negligible**: The permission regulates the access to resources or operations that have a limited impact on the device integrity. Moreover, misuses of these permissions can be easily noticed and halted by users. Examples of these permissions are the one regulating the on/off status of connectivity interfaces, such as Bluetooth, whose excessive misuse may cause battery depletion. It is worth noting that on Android devices Bluetooth activity is indicated through a status icon, which also allow the user to deactivate it.
- **Limited**: The permission regulates the access to resources, which have a limited impact on the device integrity, but whose continuous misuse can cause noticeable issues, such as battery depletion, or device misbehavior. Differently from negligible, the "Limited" index is assigned to those permissions whose (mis)use is not actively notified to the user. An example is the `CHANGE_CONFIGURATION` permission, which allows an application to modify some settings like the font-size. In the case of a misbehavior, the user would have to manually change the configuration back to the preferred value.
- **Significant**: The permission affects resources whose misuse can cause significant damage to the device. The user may not notice the effect of the misbehavior. However a full exploitation would require an association with other permissions. An example is the `BLUETOOTH_ADMIN` permission, which allows an app to handle the pairing with other devices: this may bring to the download of unrequested files on the device.
- **Relevant**: The permission regulates the access to a resource that is critical for the device integrity. The app may alter the device functionality and it may interfere with other apps. Damages caused by a misuse of these permissions could be reversed, but at cost of a significant user

effort. An example is the `SET_PROCESS_LIMIT` permission, which determines the maximum number of active processes.

- **Maximum**: The permission regulates the access to resources or operations seriously affecting the device integrity. The damage caused by misuse of this permission could be not reversible. An example is the `MOUNT_FORMAT_FILESYSTEM`, which allows an app to format removable storage, causing permanent file loss.

A list of permissions that are critical for the system threat is:

- `android.permission.INSTALL_PACKAGES`: it allows the app to install new packages. This functionality has been used by several malwares to install dangerous apps with additional permissions (e.g., the `ZFT` malware) or advertisement apps (Adware).
- `android.permission.WRITE_EXTERNAL_STORAGE`: it allows the app to modify the external memory content. An app with this permission can fill the device memory and remove or permanently modify files. As an example, the malware `Moghava` permanently damages all the pictures in the user gallery, loading instead a propaganda image.
- `android.permission.CHANGE_WIFI_STATE`: it gives to the app the control on the WiFi device status. The WiFi interface has a consistent impact on the device battery lifetime and it may cause Internet disconnection, since WiFi overrides the mobile data connection even if the access point is not connected to the Internet. Thus, a malicious control on the WiFi represents an integrity violation.

Other permissions with a consistent value of system threat are those giving access to the device interface and peripherals (e.g., camera, vibration, etc.), whose (mis)use causes both a performance or a battery overhead.

3.1.3. Financial threat score

Financial threat involves those permissions that, if misused, cause direct (or indirect) monetary loss for the user. The interpretation of the values that we have given for the financial threat is:

- **No Threat**: The permission does not imply any financial cost.

- **Negligible**: The permission regulates the access to resources whose misuse can cause limited damage in terms of monetary loss. However, the loss happens only if other conditions, not related to the permission, are verified. An example is represented by those permissions allowing the access to online resources, such as, `READ_SOCIAL_STREAMS`, which also require permissions for Internet access to impose a cost.

- **Limited**: The permission regulates the access to resources that impose a limited and indirect cost to the user. An example is the `INTERNET` permission, which allows an application to open network sockets. It can cause indirect monetary loss if the device is using the 3G/4G network to receive and send data.

- **Significant**: The permission regulates the access to resources that impose an indirect but potentially severe financial cost. An example is the `CHANGE_WIFI_STATE`, which, by deactivating the WiFi during the download of a large file, could impose the cost of the downloaded Mbytes on the 3G/4G traffic.

- **Relevant**: The permission regulates the access to a resource that may impose a direct cost to the user. This value is assigned to the `RECEIVE_SMS` permission, which controls the access to received messages, even allowing an app to intercept and hide new incoming messages. This strategy is used by SMS trojan malware to hide the registration to premium services.

- **Maximum**: The permission regulates the access to resources with a serious direct effect on user's money. An example is the `SEND_SMS` permission, which allows an application to potentially send unlimited text messages.

Some permissions that we consider critical for the financial threat are:

- `android.permission.SEND_SMS`: as introduced above, it allows an app to send text messages. With this permission, an app can virtually send unlimited messages to whatever number. Sending text messages is an operation that has a monetary cost established by the provider and that may vary with the recipient. Moreover, text messages can be used for the subscription to premium services, which impose a weekly or monthly cost. Such a strategy has been exploited by several malware known as SMS Trojan [18].

- `android.permission.CALL_PHONE`: it gives to an app the authorization to initiate phone calls. Phone calls have the same financial implications of text messages, with usually a higher cost imposed on the user [19].
- `android.permission.INTERNET`: as introduced before, it gives to an app the authorization to open sockets for external connections. Received and transmitted bytes of data are other elements that telephony providers charge to users. Opening a connection and streaming data on it always generates a cost and an app with this permission can virtually send unlimited amount of data. This permission becomes particularly dangerous if coupled with the `CHANGE_WIFI_STATE` permission discussed formerly.

3.2. Global threat score

For each app, we define the *global threat score* σ, which summarizes, in a single index, the threat scores previously introduced, the privacy, system, and financial ones. The global threat score is computed by analyzing the manifest file, and by calculating a weighted sum as follows:

$$\sigma = \frac{\sum_{i=1}^{n} w_p \cdot pt_i + w_s \cdot st_i + w_f \cdot ft_i}{1 + \lceil log(1+n) \rceil} \tag{2}$$

where n is the number of permissions declared by a specific app, pt_i, st_i, ft_i are, respectively, the privacy, system, and financial threat scores of the i-th permission required by the app. The numerator is weighed by w_p, w_s, w_f. It is worth noting how these weights can be customized to better fit the users' perception of threats. In particular, when installing the MAETROID application, users will be able to select one of three usage modes: privacy, financial or system integrity. This choice impacts on the weights, by giving a higher weight to the corresponding threat score. The weights values can also be fine-tuned by expert users. By default, in MAETROID w_f is set to be three times greater than w_s and w_p: this means that we consider the financial threat three times more relevant than the system and privacy threats. However, values and proportions among weights are not fixed and they can be adjusted, according, e.g., to the user's perception. The denominator of Equation (2) has been introduced so that the threat represented by requested permissions is considered more relevant than the number of permissions. We consider apps with σ lower than 4 as *low-threat* apps, while apps with σ between 4 and 7 ($4 \leq \sigma < 7$) are considered *moderate threat* to *high-threat*. Higher values of σ ($\sigma \geq 7$) mean *extremely* critical apps.

Summarizing, the value σ estimates how much an app is critical from the security point of view, by considering the declared permissions only. Hence, the more permissions are required by an app, and the more dangerous these permissions are, the more critical the app becomes. The fact that an app receives a low global threat score should increase the likelihood that this app will be downloaded and, as a consequence, this should encourage developers to accurately choose the permissions required by their apps.

3.2.1. Criteria and their association with the application labels

In addition to the global threat score σ, in the following we present the other criteria that will be used to rank the trustworthiness of an Android application.

Market (μ). Apps are normally distributed through online marketplaces. The most popular market is Google Play, also referred as the *official* market. This market is considered a more protected environment than unofficial ones since app developers build their reputation on the base of the apps they publish. More specifically, a developer, who wants to publish apps on Google Play has to buy a developer account, at the price of 25$. In exchange, the developer receives a private key, which she can use to digitally sign her apps [20]. If users report an app as malicious, then this app is removed both from the market and remotely from all the devices that have installed it. Moreover, the developer can be tracked and blacklisted. In addition, Google Play includes some reputation indexes that should help users to understand the app quality. These features make the official market a trustworthy place where to download apps. Nevertheless, several malware have also been found in the official market [3] [21] [22].

There also exists a plethora of *unofficial* marketplaces, among which:

- `http://www.appbrain.com`,
- `http://www.aptoide.com`,

- `http://www.androiddrawer.com`,
- `http://androidlife.ru`,
- `http://www.anruan.com`,
- `http://www.appsapk.com`,
- `http://android.pandaapp.com`,
- `http://slideme.org`.

These marketplaces do not require developer registration and they still attract several users since they usually give access to apps that are not available on the official market, or they distribute free versions of apps, which should be bought instead on the Play store. However, unofficial markets often miss reputation indexes and, sometimes, there is no control at all on the quality of the apps, so it is easier to download malicious apps. Related to the market source, in the problem instantiation we also consider another label, namely apps that are *manually installed*, which happens when the user manually installs the app (e.g., when she downloads the app `apk` and installs it through a file manager).

Thus, given the three possible values featured by an app with respect to its market (official, unofficial, and manually installed), we define the relative *relevance* of the three labels MAETROID assigns to the application as follows:

- $\mu =$ official: we consider that *Trusted* is moderately more relevant than *Medium Risk* and strongly more relevant than *High Risk*;
- $\mu =$ unofficial: we consider that *High risk* is moderately more relevant than *Trusted* and slightly more relevant than *Medium Risk*;
- $\mu =$ manually installed: we consider that *High Risk* is slightly more relevant than *Trusted* and *Medium Risk* (that are equally relevant).

According to this information, it is possible to fill the AHP comparison matrix expressing the relevance of the alternatives towards the criterion Market.

Developer (δ). We consider three types of developers: *Standard*, *Top*, and *Google*. Google rewards the best app developers with a *Top Developer* badge, which is reported on each app they publish. Hence, these developers should be considered strongly trusted since they usually produce high-quality apps and they should not be interested in lowering their reputation. On Google Play, *Google Inc.* itself is considered a Top Developer. However, we consider Google more trusted than other developers, given the interest that Google has in protecting the security of Android users. It is worth noting that such an assumption is in line with the trust hypothesis for Signature and Signature-or-System permissions. All the other developers are considered standard and, since the Top Developer badge is only used on Google Play, all developers of apps coming from unofficial markets have been labeled standard as well, since on these markets apps are not digitally signed. The AHP comparison matrix for the developer parameter is defined according to the following analysis:

- δ = Google: we consider that *Trusted* is extremely more relevant than *Medium Risk* and *High Risk* (that are equally relevant);
- δ = Top Developer: we consider that *Trusted* is very strongly more relevant than *High Risk* and *Medium Risk* (that are equally relevant);
- δ = Standard: we consider that the three alternatives are equally relevant.

User rating (ρ). On several markets, users can rate apps and leave a comment, which can be shown to other users. Rating is generally expressed as a number that ranges from 1 to 5 (or it is normalized in this range). We consider apps with a rate less than 2 as *low-quality*, for which the *Medium Risk* alternative is extremely more relevant than the *Trusted* one. A score higher than 4 means a *high-to-very-high* quality apps for which the *Trusted* alternative is very strongly more relevant than the others. Intermediate values mean a neutral comparison matrix.

Number of Downloads (η). Several markets report the number of downloads for each app. As an example, the so-called "killer apps", i.e., extremely popular apps, have been downloaded from Google Play more than 100 millions of times. These apps should be considered differently from those downloaded a low number of times, e.g., less than 100 times. In fact, apps with a very high amount of downloads are popular apps already tested by several users and are, usually, more trustworthy. Note that the number of downloads, though independent, is needed to contextualize the User Rating criterion: a rating of five stars (out of five), given to an app downloaded by a single user, is indeed not very relevant. Hence, we define 7 intervals in which the value η may fall. For very high values of η, *Trusted* is extremely relevant. As the value of η decreases, the relevance gradually turns from *Trusted* to *High Risk*.

Global threat score (σ). As explained in Section 3.1, a global threat score is associated to each app. According to its value, the relative relevance of the three possible application labels is as follows:

- $\sigma < 4$: *Trusted* is very strongly more relevant than *High Risk* and moderately more relevant than *Medium Risk*;
- $4 \leq \sigma \leq 7$: *High risk* is very strongly more relevant than the other alternatives (that are equally relevant);
- $\sigma > 7$: *High Risk* is extremely more relevant than *Trusted*, and *Medium Risk* is strongly more relevant than *Trusted*.

On the comparison matrices. For those marketplaces without download counters and/or rating systems, we have defined two additional comparison matrices whose elements are all equal to 1. In such a way, when using these matrices to evaluate the relevance of the alternatives with respect to the criterion, the alternatives have the same relevance for that criterion. Hence, this criterion will not influence the decision.

We have defined 20 comparison matrices, but it is possible to increase their number to have a finer, or customized, granularity for each criterion. Finally, it is worth noting that the list of proposed criteria is not exhaustive, and the methodology enable the insertion of other criteria to evaluate the alternatives.

Finally, in the current implementation, we consider all the criteria as equally relevant in achieving the goal. However, we highlight that the relative importance among the criteria, with respect to the goal of classifying an application, can be customized, as to have more granular results with respect to the user's expectation. As an example, a user can decide to give more importance to the developer criterion rather than to the market one. This could happen in those situations in which the user always downloads from official markets. In the AHP-based MAETROID design, this would lead to a comparison matrix where the developer criterion is, e.g., very strongly more relevant than the market one.

3.3. Classification

The classification process of MAETROID is depicted in Figure 2. The system takes as input an Android application, which belongs to one of three possible classes:

- **Good Apps:** applications behaving as expected by the user, well designed and bugs-free, generally popular and with good rating, which also do not contain malicious code;
- **Bad Apps:** applications that, even if they do not contain malicious code, do not behave as users expect, e.g., they are poorly designed, malfunctioning (bugs, crash, missing functionalities), or they have a high impact on performance;
- **Infected Apps:** applications that contain malicious code, specifically designed to harm the user and/or the device.

As output, we have the same app, labeled with one of the three indexes of the right-hand side of the figure (Trusted, Medium Risk, High Risk). Ideally, the classification is correct if a Good App is classified as Trusted, a Bad App is classified as Medium Risk and an Infected App is classified as High Risk. The criteria used to perform the classification are the market reputation, the developer reputation, the global threat score, the downloads number, and the user rating.

4. MAETROID Implementation and Results

The MAETROID framework has been implemented as an app for Android devices. The MAETROID app is composed of an activity[5] and several services[6] running in background. Whenever a new app is being installed, MAETROID intercepts the event broadcast by Android through an intent

[5]`http://developer.android.com/reference/android/app/Activity.html`

[6]`http://developer.android.com/guide/components/services.html`

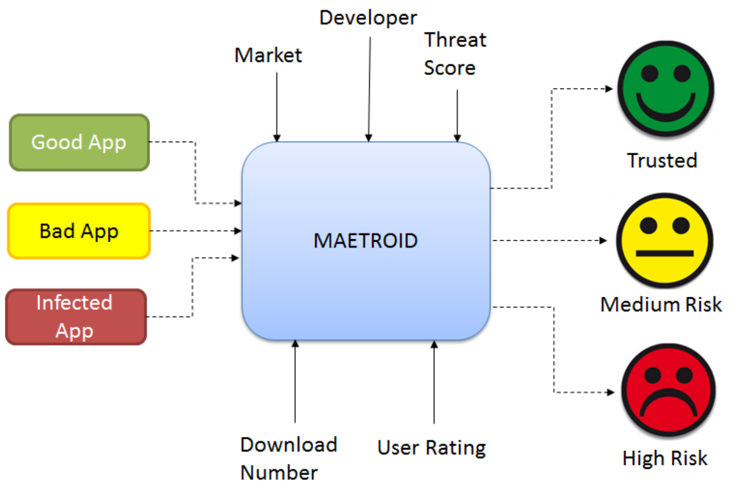


Figure 2: MAETROID Classification Process

filter[7]. Hence, MAETROID comes in foreground with its activity, showing that the app is being analyzed to the user. In the meanwhile, the analysis services transparently retrieve the values of the five criteria (market, developer reputation, user rating, number of downloads and global threat score) for the app under investigation. The global threat score is computed by parsing the app manifest file, to retrieve the set of requested permissions, and by computing the global threat score as shown in Section 3.2. Note that the evaluation is performed locally on the user device, whenever a new app is going to be installed. Hence, there are not scalability issues on the market, since the market is not affected by the computation.

The market is inferred from the installer of the downloaded app. In fact, as discussed in Section 3, both official and unofficial markets provide an on-device custom app called "installer", which is used to browse the marketplace, to download and install some selected apps. The name of the installer is reported in the message which is broadcast by the OS, to communicate the event of a new app installed on the device. The other criteria (user rating, number of downloads and developer reputation) are extracted by parsing the market's web page. Upon computing the values for the five criteria,

[7]http://developer.android.com/guide/components/intents-filters.html

MAETROID implements AHP through `Jama`, the Java matrix package for matrix calculi[8].

Upon completing the analysis, MAETROID returns its decision in the form of a smiley. We have decided to use such a simple graphical representation to make the outcome more user-friendly and understandable. If the app is considered High Risk (resp., Medium Risk), the user is advised of the potential threat through a red 'sad' smiley (resp., a yellow 'poker face' smiley), and she is asked if she wants to uninstall the app. If the user decides to uninstall the app, MAETROID handles the uninstallation process. Otherwise, if MAETROID considers the app as trusted, the user is invited through a green 'happy' smiley to run the installed app.

It is worth noting that MAETROID does not block the installation process, but it prevents the app from being started until the analysis outcome is shown and the user has taken her decision. The fact that installation process is not blocked a priori by MAETROID is not harmful. In fact, in Android, an app does not perform any operation (including deploying assets and saving files on the device) until it runs. Given that users follow the MAETROID advise, a dangerous app can neither harm the user nor device, since apps can be opened only after that users trigger the app start from the `launcher` app. A key point in MAETROID is giving the users a "peace of mind". Indeed, Android users will have little or no interaction with the classification process and they will be relieved from the burden of understanding the security implications brought both by the values of the five criteria and by the permissions list.

We remark that MAETROID performs the classification directly on the device. This approach has the following advantages:

- MAETROID is not affected by scalability issues, since it is not necessary to classify *a priori* all the existing Android apps, building a centralized database which would need continuous update and maintenance;

- app updates automatically trigger a reclassification process when the updated version is downloaded. Thus, if the classification result changes because the updated version asks for more permissions, the new version will not run on the device, given that the user follows the MAETROID

[8]`http://math.nist.gov/javanumerics/jama/`

decision.

The whole sequence of steps of the MAETROID analysis process is depicted in Figure 3 and it is summarized as follows:

Step 1: a new app is downloaded locally from the marketplace;

Step 2: the user decides to perform the app installation;

Step 3: the installation process is hijacked (and paused) by MAETROID (which was previously installed on the user device);

Step 4: MAETROID retrieves the metadata used to perform the classification, locally from the app manifest file and remotely from the marketplace;

Step 5: MAETROID exploits the retrieved metadata to apply the AHP classification, locally on the device;

Step 6: the decision is shown to the user, in form of a smiley;

Step 7: the user decides whether to continue the installation or remove the app, based on the output of the classification.

4.1. Classification Results

To evaluate the MAETROID performances, we have conducted two sets of experiments, one on a large set of applications coming from known databases and one on a smaller set, manually analyzed.

In the first set, we have used the classification algorithm of MAETROID to classify a dataset of 11,046 apps. This dataset is composed of 9,804 apps selected from the official market Google Play, while the remaining 1,242 apps come from the database of known malware Genome[7]. To extract the metadata of the Google Play apps, we have built a crawler to retrieve the rating, download number and permissions set, starting from an existing database[9].

The classification results are shown in Figure 4 and also reported in Table 3 for the sake of clarity. As shown, malicious apps of the Genome database

[9]The crawler is based on: `https://github.com/MarcelloLins/GooglePlayAppsCrawler`

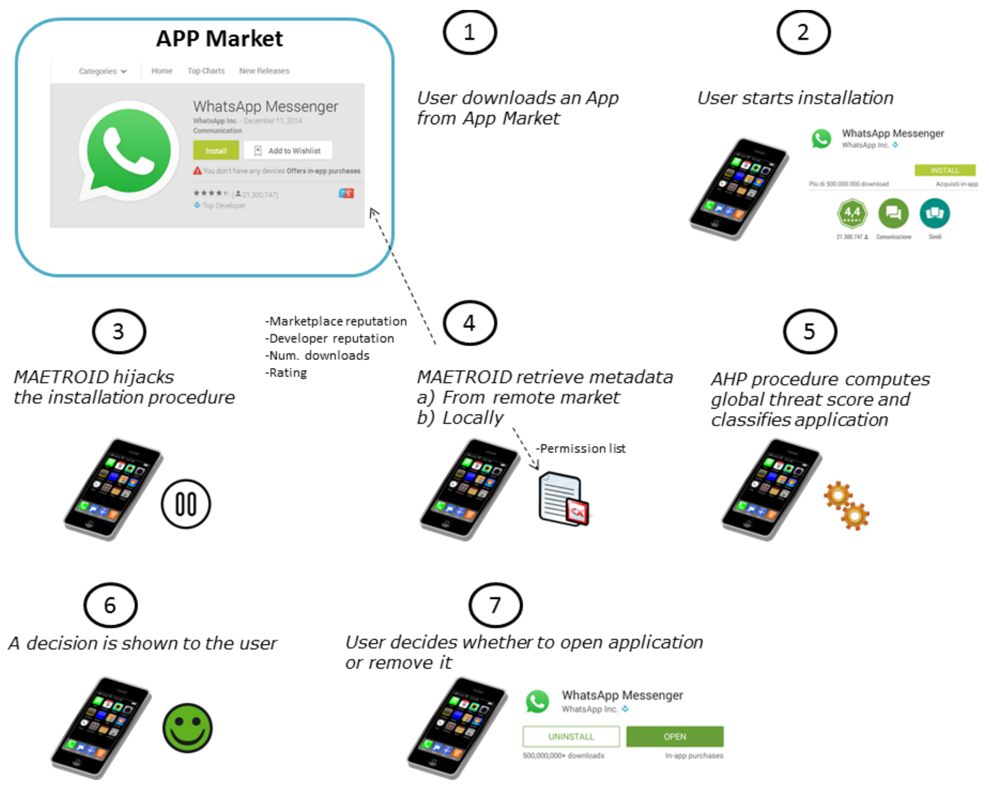

APP Market
Categories Home Top Charts New Releases
WhatsApp Messenger
Install
Add to Wishlist
1
User downloads an App from App Market
2
User starts installation
WhatsApp Messenger
INSTALL
3
MAETROID hijacks the installation procedure
-Marketplace reputation
-Developer reputation
-Num. downloads
-Rating
4
MAETROID retrieve metadata
a) From remote market
b) Locally
-Permission list
5
AHP procedure computes global threat score and classifies application
6
A decision is shown to the user
7
User decides whether to open application or remove it
WhatsApp Messenger
UNINSTALL
OPEN

Figure 3: Overview of MAETROID Process

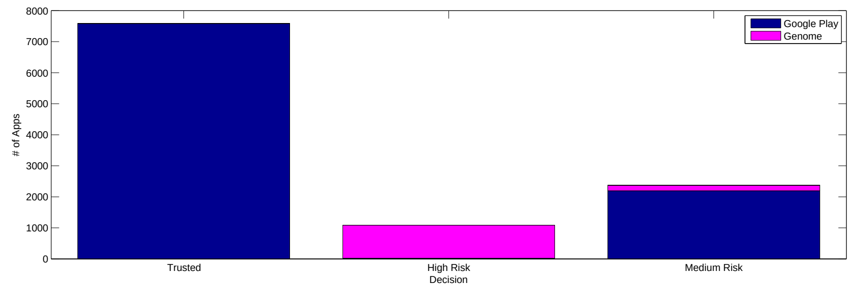


Figure 4: Classification Results on 11,046 Apps

have all been classified as risky by MAETROID. More precisely, 85% of the malicious apps of Genome have been classified as High Risk and the remaining 15% as Medium Risk. None of the apps from Genome have been classified as Trusted. We have used the apps from Google Play as a control set. We can see that the greatest share of Play apps (77.37%) have been classified as Trusted, while 22.4% have been classified as Medium-Risk and only 0.23% as High-Risk. It is worth noting that, in this test, we have no previous knowledge about the nature of the apps from Google Play. However, we argue that the classification results over the control set are plausible. In fact, the results show that 77.37% of the apps from Google Play do not represent a threat to security, while about 22% of the apps show some criticality, usually due to a low number of downloads. Only a very small number of apps represent a potential threat to security, mainly due to the set of dangerous permissions they ask[10]

Table 3: Classification results on the first set (11,046 apps)

	Trusted	**High Risk**	**Medium Risk**
Google Play	7586	22	2196
Genome	0	1064	178

In the second set of experiments, we have verified the classification accuracy of MAETROID, by measuring both its precision and recall, i.e., the overall classification error. The testbed dataset does not overlap with the previous one and it is composed of 180 Android apps, which are known in advance to be:

- *Safe Apps*: those apps that behave correctly both from the security and functional point of view. This class is further divided in two subclasses: *Official* and *Unofficial*, stating, respectively, whether the app has been downloaded from Google Play or not. Good Apps are correctly classified by MAETROID if its output is "Trusted" (green happy smiley);

- *Apps with Unwanted Behavior*: the app permissions given to these apps may be used to cause potentially unwanted behavior, such as with Adware. These apps are correctly classified by MAETROID if its output is "Medium Risk" (yellow poker face smiley);

[10]Additional details on this set of experiments, i.e., links to the tested apps and their metadata, can be retrieved at `http://www.android-security.it/maetroid/app_list_final.xlsx`.

- *Malicious Apps*: those apps infected by a malware. Malicious Apps are correctly classified by MAETROID if its output is “High Risk” (red sad smiley).

In more details, the test-set consists of 180 manually analyzed apps: 90 from Google Play, 50 from unofficial markets, and 40 manually installed. Among all these 180 apps, 40 were infected by well-known malware. Apps belong to different categories: augmented reality, books and news, communication, desktop manager, entertainment, file managers, game, social and utility, and anti-virus. The app user rating ranges over $[1, 5]$, the number of downloads ranges over [0, 10M+], and the apps were produced either by standard developers, or by Top Developers, or by Google.

The MAETROID outcome over the 180 apps set is reported in Figure 5, where the x-axis shows the three possible output labels of MAETROID (Trusted, High Risk, Medium Risk), while the y-axis shows the number of apps classified per outcome. The light-green colour represents safe apps (in dark green the ones coming from unofficial markets), the red color represents apps infected with malware, whilst violet (vertical lines pattern) represents apps with unwanted behavior. All the infected apps have been correctly recognized by MAETROID as *High Risk*. It is worth noting that some good apps also fall in this class. These apps come from unofficial markets (labeled as “Good Apps (Unoff.)” in Figure 5). Since no user rating is available for these apps, MAETROID applies a safe approach by considering them as High Risk, at least initially. However, as soon as new information become available for these apps [23], they will eventually be classified as trusted ones.

All the apps coming from Google Play have been classified either as *Trusted* or *Medium-Risk*, based upon user rating, global threat score, and number of downloads. All the apps with unwanted behavior coming from Google Play have been correctly considered as *Medium-Risk*. These apps do not work as expected or they crash upon starting. An example of this class of apps is the game `Avoid the Ghosts`[11], which is a reproduction of the classic Pac Man game. The app does not work correctly: in fact, when the app starts, it is impossible to control Pac Man movements. The app has been found on the official market, but it has been downloaded few times and it has received bad ratings. However, `Avoid the Ghosts` does not require

[11]The app was available on Google Play at time of our experimentation, while at time of writing it was no more available.

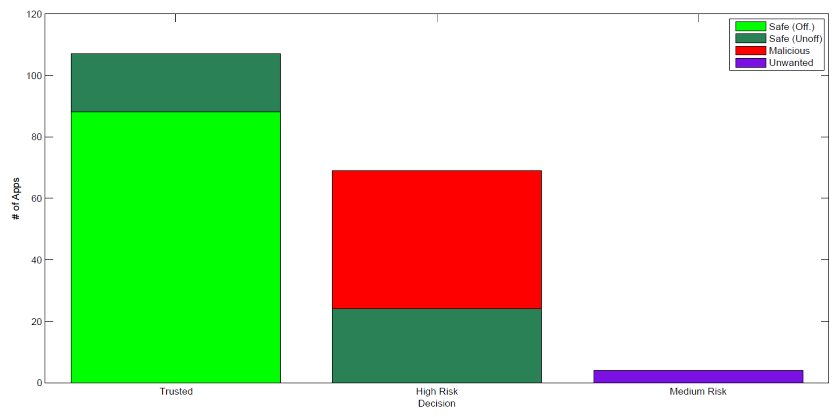


Figure 5: Classification results on the second set (180 apps)

dangerous permissions and, hence, it is considered *Medium Risk* instead of *High Risk*.

In the following, we detail the classification process for two popular apps, namely Angry Birds Space and Skype.

Classification example 1: Angry Birds Space. The values of the five criteria computed by MAETROID are shown in Table 4. In detail, the app developer is a Top Developer, the app has been downloaded by more than 10 millions of users, receiving a global rating of 4.7. Furthermore, it comes from the official market Google Play and it has a low global threat score (2.7).

Table 4: Values of criteria for Angry Birds Space

σ	ρ	μ	δ	η
2.7	4.7	Google Play	Top Developer	10M+

Table 5 shows the matrix used to compare the three alternatives, with respect to the "App Developer" criterion. Top Developers generally produce high quality apps and they are not likely to publish malicious apps. According to this intuition, we assigned the following pairwise relevance to the alternatives: Trusted is very strongly favorite with respect to Medium Risk and strongly favorite with respect to High Risk. Trusted (green happy smiley) obtains the highest priority (0.7) compared to the other two alternatives.

MAETROID merges the local priorities for criterion "developer" with the ones coming from the comparison matrices of the other four criteria. We

Table 5: Comparison matrix for Top Developer for `Angry Birds Space`

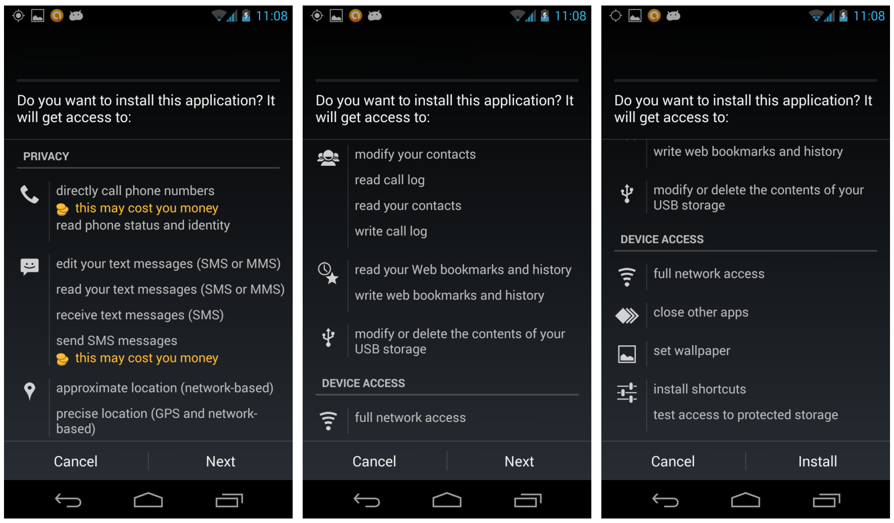
Top Developer	Trusted	High Risk	Medium Risk	Local Priorities
Trusted	1	4	7	0.7
High-Risk	$\frac{1}{4}$	1	4	0.23
Medium-Risk	$\frac{1}{7}$	$\frac{1}{4}$	1	0.07

finally obtain the following global priorities $[0.7, 0.16, 0.14]$. The three values represent the priorities for the three alternatives, respectively Trusted, High Risk, and Medium Risk. Trusted is the alternative with the highest value and, thus, it is also the result of the MAETROID classification for that app.

On the contrary, when MAETROID analyzes a version of `Angry Birds Space` found on a database of infected apps, it outputs High-Risk as the highest priority. This app has been found in the past to be infected by the malware `Geinimi` [18]. The malware steals information concerning both the user and the device, which are sent via SMS to a number controlled by the attacker. To perform these further operations, the malware asks for several other permissions (Figure 6), leading to a global threat score equal to 7.3. This high value for the global threat score correctly drives MAETROID towards the High-Risk outcome.

Classification example 2: Skype. `Skype` is a popular software used for VoIP and free chat and its mobile version is considered reputable, since it enables phone calls with smartphones, using the data connection instead of traditional (and more expensive) landline and cellular calls connections. To work properly, the Android version of `Skype` requires a large number of permissions. Computing the global threat score by means of Equation (2), `Skype` gets a score of 6.8. Skype is an example of a high-threat app. In our analysis, we have considered two Skype versions, one from the official market and the other from an unofficial market, see Table 6.

The global priorities vector computed by MAETROID on the version coming from the official market is $[0.47, 0.4, 0.13]$. This slightly favors the Trusted alternative. Both the marketplace and the large number of downloads increase the trustworthiness of this app, even if it has a high global threat score. For the version from the unofficial market, which does not even provide a download counter, the global priorities are very different from the



11:08

Do you want to install this application? It will get access to:

PRIVACY

directly call phone numbers
this may cost you money
read phone status and identity

edit your text messages (SMS or MMS)
read your text messages (SMS or MMS)
receive text messages (SMS)
send SMS messages
this may cost you money

approximate location (network-based)
precise location (GPS and network-based)

Cancel | Next

11:08

Do you want to install this application? It will get access to:

modify your contacts
read call log
read your contacts
write call log

read your Web bookmarks and history
write web bookmarks and history

modify or delete the contents of your USB storage

DEVICE ACCESS

full network access

Cancel | Next

11:08

Do you want to install this application? It will get access to:

write web bookmarks and history

modify or delete the contents of your USB storage

DEVICE ACCESS

full network access

close other apps

set wallpaper

install shortcuts
test access to protected storage

Cancel | Install

Figure 6: Permissions declared by `Angry Bird Space` trojanized by Geinimi

Table 6: Two Skype versions

Name	σ	ρ	μ	δ	η
`Skype_1`	6.8	3.8	Google Play	Standard Developer	10M+
`Skype_2`	6.8	4	Unofficial	Standard Developer	N.A.

previous ones: $[0.29, 0.52, 0.19]$ and the app is labeled as High Risk ("sad" smiley). Even if the two versions require the same set of permissions, it is possible that their source codes are different (possibly malicious). Since more than 10 millions of users have downloaded the version from the official market, it is strongly unlikely that malicious behaviors have not been noticed and reported, forcing the app removal from that market.

5. Discussion

MAETROID is an app that helps users to understand the level of risk of downloaded apps, i.e., if those apps have potential security and privacy risks. Several factors contribute to make an app likely dangerous. The most relevant is the requested set of permissions, which effectively gives the app the power to invoke critical functions and access critical resources. This, *per se*, cannot be the only criterion to decide whether an app is risky or not. In fact, genuine apps may request these permissions to legally access, for example, the contact detail (as a contact manager), or to send SMSs (as a customized text manager), and so on. Obviously, if these permissions were dangerous regardless of the apps, then Android would remove the possibility of using them. Hence, other criteria can be relevant, such as the marketplace (an official market has usually a pre-filtering step to remove some malicious or repackaged apps), the developers (the most important ones do not want to risk their reputation), the user ratings, and the number of downloads.

MAETROID is shipped as a custom app to run on the user-device. Whenever a new app is downloaded, the installation process is paused to run the classification process locally. Then, MAETROID gives the user a friendly decision about the app risk level, and it is up to the user to decide whether the installation needs to be resumed or terminated. We have to point out that MAETROID is not an anti-virus nor an intrusion detection system. MAETROID focuses on elements which are visible to each user, i.e., permissions, market and app popularity, it evaluates them and then generates a single, easy to understand, decision. Furthermore, it does not analyze the app's code. On the contrary, anti-virus solutions base their decision by looking statically for known bad signatures inside the app's code (black-list approach). Other approaches, such as anomaly intrusion detection systems, look for anomaly patterns at run-time. For these reasons, we see MAETROID as an orthogonal approach to these solutions. In particular, MAETROID can be the first line of defense in deciding whether an app may be potentially dangerous. As an example, if an app is classified as Medium Risk (yellow face), one can decide to run it in a sand-boxed environment under control of an intrusion detection system.

As introduced in the previous sections, MAETROID acts at deploy-time and no further checks are performed after that the user accepts the decision. In particular, MAETROID does not enforce security at run-time, to not impose any overhead on the apps execution. The approach of MAETROID is also scalable, since apps are evaluated at deploy-time, directly on the user device. Thus, despite the huge amount of available Android apps and their possible different provenances, the proposed approach is viable, since (i) it simply requires the users to install the MAETROID app on their devices (ii) apps are evaluated at installation time only.

We have analyzed the user response and acceptance of MAETROID, designing and proposing a survey to a set of about 200 subjects at a public event about Internet technology[12]. The survey's results have been analyzed to synthesize outcomes on the users perception on mobile security and related threats and to receive a feedback on the MAETROID effectiveness and usability. The results of the survey show that the interviewees are aware of mobile security threats, since only 10% of them state that they are not concerned about possible threats at all. However, more than 25% of the subjects gives no importance to the Android security warnings and only 27% of the subjects considers the Android permissions as useful and meaningful. The results of the survey have also confirmed that the MAETROID evaluation is effective in driving the users decision, by avoiding the installation of malicious apps. In particular, 90% of the interviewees changed their mind about installing a malicious app after that MAETROID evaluates the app as dangerous.

Since MAETROID uses a single index for evaluation, with three possible values only, this might somehow be too coarse-grained in some situations. As an example, the vast majority of apps found on official market, such as Google Play, are classified as trusted. This is because it rarely happens that malicious apps are hosted on official markets. However, one could argue that, even if safe, these apps sometimes require a too large set of permissions and this should be pointed out. Further, advanced users would like to know more about the threats for each of the MAETROID categories (financial, privacy and system), which is not represented by the final output. The first issue can be easily taken into account by penalizing apps that requires a large number of permissions, regardless of the market, the developer and the number of download. As we have already detailed, the parameters, such as the weight given to each criteria, can be customized by the users. Regarding the second issue, one future development concerns the addition of some further explanations to the final decision, in the form, e.g., of an optional tab that the user can open to understand in detail the final decision.

6. Related Work

Several extensions and improvements to the Android permission system have been recently proposed. The work presented in [24] proposes a security

[12]`http://www.internetfestival.it/`

framework that regulates the actions of Android apps defining security rules concerning permissions and sequence of operations. New rules can be added using a specification language. The app code is analyzed at deploy-time to verify whether it is compliant to the rule, otherwise it is considered as malicious code. With respect to this work, MAETROID does not require the code to be decompiled and analyzed. Indeed, it only requires the permissions list that can be retrieved from the manifest file and other pieces of information that can be retrieved from the website where the app can be downloaded.

Authors of [10] present a finer grained model of the Android permission system. They propose a framework, named *TISSA*, that modifies the Android system to allow the user to choose the set of permissions she wants to grant to an app and those that have to be denied. Using mocking data, they ensure that an app works correctly even if it is not allowed to access the required information. However, their system focuses on the analysis of privacy threatening permissions and it relies on the user expertise and knowledge. A work similar to TISSA is presented in [25], where the authors design an improved app installer that allows users to define three different policies for each permission: allow, deny, or conditional allow. Conditional allow is used to define a customized policy for a specific permission by means of a policy definition language. However, the responsibility of choosing the right permissions still falls on the user, whilst MAETROID directly shows to the user the risk classification of the app, performing automatically the permissions analysis.

In [26] and [27], the authors present a multi-level behavior-based intrusion detection system called MADAM. The proposed system learns the correct devices' behavior and then detects significant deviations signaling an intrusion. The MADAM approach is orthogonal to that of MAETROID because MADAM analyzes the app behavior at run-time, while MAETROID performs a risk analysis before installing the app. In [28], apps have been classified based on their required permissions. Apps have been divided in functional clusters by means of Self Organizing Maps, proving that apps with the same set of permission have similar functionalities. However this work does not differentiate between good and bad (trojanized) apps. Another analysis of Android permissions is presented in [12], where the authors discuss a tool named *Stowaway*, which discovers permission over-declaration errors in apps. Using this tool, it is possible to analyze the 85% of Android available functions, including the private ones, to obtain a mapping between functions and permissions. This work mainly concerns the analysis of permissions without proposing a direct link between declared permissions and apps security, as with MAETROID. A system to implement security policies on Android devices is presented in [29]. This system is based on the introduction of a monitor of security critical functionalities, which matches the performed actions with security policies defined by the mobile device user. However, the presence of the monitor imposes a *consistent* overhead. Another security framework based on user defined policies, preventing app to perform non compliant operations, is presented in [30]. The framework attempts to reduce the overhead and to improve the effectiveness through a probabilistic contract based approach. This leads to a probabilistic satisfaction of security requirements.

`TrustGo` [31] is a framework aimed at classifying mobile apps exploiting a multi-criteria analysis. `TrustGo` gives users a full description of the security threats brought by an app and also works as an antivirus. `TrustGo` is catalog-based: available Android apps are analyzed by security experts and are inserted in a catalog, checked when a `TrustGo` user is installing a new app. `TrustGo` is effective and the catalog-based approach ensures a good accuracy. However, it is not possible to collect all the existing apps, since only the apps distributed through official channels can be analyzed. Moreover, if an app is updated, it is possible that some security features may change in the new version, i.e., new permissions are added in the manifest and this requires a catalog update, which may not be triggered in time. On the other hand, MAETROID is independent from the app version, i.e., the app is analyzed “as is”. App update will trigger a new classification process. Moreover, MAETROID classifies the app at deploy-time, without requiring any centralized catalog. Thus, any app can be classified even if coming from unknown marketplaces. Another app classification system is presented in [32], where apps are classified in comparison with formerly analyzed apps. The methodology exploits probabilistic generative models to analyze apps on different criteria including permissions. However, the performed analysis is more effective in creating an awareness on developers in trying to avoid issues like permission over-declaration, instead of providing an index effective in driving the user decision on the app installation.

Analysis of the Android permission understanding have been performed in [11], where subjects from an university campus have been asked to fill a survey on Android security and on their current approach to the permission security mechanism. Recently, Android has introduced a service of remote monitoring of installed apps, called *VerifyApps* [33], which acts as a remote antivirus by looking directly for known malware signature. The visual approach is similar to MAETROID: when an app is considered dangerous, the user is advised about the potential threat and asked if she desires to install the app considered dangerous. On the contrary, the risk analysis of MAETROID is based on different parameters that do not depend from the app code. A dangerous app can be considered malicious by MAETROID even if it is a brand new app with an unknown signature.

A similar approach to MAETROID is Androlyzer [34], which is a web-based service that gives the user a lot of information about the used API, used libraries, privacy leaks, requested permission, which might be too overwhelming for an ordinary user. For these reasons, in MAETROID we have decided to keep the output of the results as simple as possible. This is a first step towards a better understanding of the risk of an app from the point of average users. Furthermore, these reputation services, again with similar one in [35], are usually centralized, hence they are not very scalable. In fact, these services need, first of all, to download all the apps (or the most important ones), and are usually limited to unofficial markets. Furthermore, their databases need to be constantly updated and the centralized service need to cope with several concurrent requests of different users. On the contrary, MAETROID is run locally on the user device and only for the newly downloaded app so there are not scalability issues due to checking a large number of apps concurrently.

Finally, MAETROID strongly differs, by design choice, from security frameworks like MOCANA [36] or Samsung Knox [37] which are designed for "high-security government or military deployment", enforcing security from the hardware to app level through trusted storage for remote attestation and a dedicated market where only vetted apps are published. Furthermore, the configuration of these systems are usually centralized and implemented by expert administrators. The MAETROID solution is designed for a wider set of users, not requiring dedicated hardware nor customized OS, and with little (or none) knowledge of security.

7. Conclusions

Protecting users from dangerous apps is a compelling issue. Though the main mobile OSs have already introduced some security mechanisms for device and user protection, these still present several usability-related flaws. Normally, users have a realistic view of mobile security threats and are willing to protect their devices. However, users are often tricked and they install malicious apps looking as genuine. This mainly happens since they tend to consider the app popularity and user rating more important than, e.g., the declared permissions. To this end, we have developed MAETROID, a multi-criteria decision framework for the analysis of Android apps. MAETROID has been exploited to classify more than 11,000 Android apps, coming from Google Play, unofficial markets, and databases of known malware. In our experiments, the trustworthiness index of MAETROID has proved to be able to drive correct decisions on whether to install dangerous apps.

We believe that the introduction of a simple index, as the one in outcome by MAETROID, may improve the overall mobile device security and the user awareness. In fact, suspicious apps could be identified and further analyzed, before being executed by users. Moreover, the presence of the global threat score could be an incentive for developers to accurately choose the permissions needed by their apps, effectively tackling also the permission over-declaration issue. Finally, MAETROID comes as an Android app which enforces security without imposing overhead to the user, since it becomes active only when installing a new app.

Acknowledgement

The research leading to these results has received funding from the EU Seventh Framework Programme (FP7/2007-2013) under grant no 610853 (COCO Cloud), Regione Toscana (POR CReO 2007-2013) line of activity 1.5.a - 1.6 (SECURE!) and My Information Bubble (Funded by Registro.it).

References

[1] Global mobile statistics 2014 Part B: Mobile Web mobile broadband penetration; 3G/4G subscribers and networks., Last access, 4th Dec., 2015 (2014).
URL `http://goo.gl/gJD8oA`

[2] IDC Corporate USA, Worldwide Quarterly Mobile Phone Tracker., Last access, 4th of Dec., 2015 (2013).
URL `http://goo.gl/NxMZw`

[3] FSecure Mobile Threat Report Q1 2014, Last access, 4th of Dec., 2015.
URL `http://goo.gl/AEEFZw`

[4] Pulse Secure, Mobile Threat Report, Last access, 4th Dec., 2015 (2015).
URL `https://goo.gl/LiZI96`

[5] PandaLabs quarterly report January-March 2014, Last access, 4th of Dec, 2015 (2014).
URL `http://goo.gl/hq4V0Z`

[6] N. Viennot, E. Garcia, J. Nieh, A Measurement Study of Google Play, SIGMETRICS Perform. Eval. Rev. 42 (1) (2014) 221–233.

[7] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 95–109.

[8] G. Dini, F. Martinelli, I. Matteucci, M. Petrocchi, A. Saracino, D. Sgandurra, A Multi-criteria-Based Evaluation of Android Applications, in: Trusted Systems, Vol. 7711 of LNCS, Springer Berlin Heidelberg, 2012, pp. 67–82.

[9] J. Forristal, One root to own them all, Last access, 4th of Dec., 2015 (2013).
URL `https://goo.gl/BTxNIu`

[10] Y. Zhou, X. Zhang, X. Jiang, V. W. Freeh, Taming information-stealing smartphone applications (on android), in: 4th International Conference on Trust and Trustworthy Computing (TRUST 2011), 2011, pp. 93–107.

[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, D. Wagner, Android permissions: User attention, comprehension, and behavior, in: Proceedings of the Eighth Symposium on Usable Privacy and Security, SOUPS '12, ACM, New York, NY, USA, 2012, pp. 3:1–3:14.

[12] A. P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android Permissions Demystified, in: ACM (Ed.), 8th ACM conference on Computer and Communications Security (CCS'11), 2011, pp. 627 – 638.

[13] T. L. Saaty, Decision-making with the AHP: Why is the principal eigenvector necessary, European Journal of Operational Research 145 (1) (2003) 85–91.

[14] T. L. Saaty, How to make a decision: The analytic hierarchy process, European Journal of Operational Research 48 (1) (1990) 9–26.

[15] CNIL, How to implement the data protection act, Last access, 4th of Dec., 2015 (2012).
URL `http://goo.gl/jdlw5O`

[16] G. Dini, F. Martinelli, I. Matteucci, M. Petrocchi, A. Saracino, D. Sgandurra, A Multi-Criteria-Based Evaluation of Android Applications, Tech. rep., Istituto di Informatica e Telematica, CNR, Pisa, TR-13-2012. (2012).

[17] The Organisation for Economic Co-operation and Development (OECD), The OECD Privacy Framework, Last access: 10th of December, 2015 (2013).
URL `http://goo.gl/Jy9mg2`

[18] Xuxian Jiang, Multiple Security Alerts: New Android Malware Found in Official and Alternative Android Markets, Last access, 4th of Dec., 2015 (2011).
URL `http://goo.gl/TwKu7Z`

[19] 900 pay-per-call and other information services, Last access, 4th of Dec., 2015 (2014).
URL `https://goo.gl/g2eM6Y`

[20] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, B. Shastry, Practical and Lightweight Domain Isolation on Android, in: ACM (Ed.), 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM11), 2011, pp. 51 – 61.

[21] A. P. Felt, M. Finifter, E. Chin, S. Hanna, D. Wagner, A survey of mobile malware in the wild, in: ACM (Ed.), 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM11), 2011, pp. 3 – 14.

[22] R. Cannings, An update on Android Market security, Last access, 4th of Dec., 2015 (2011).
URL `http://goo.gl/pUgtd`

[23] G. Dini, F. Martinelli, I. Matteucci, A. Saracino, D. Sgandurra, Evaluating the Trust of Android Applications through an Adaptive and Distributed Multi-criteria Approach, in: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2013., 2013, pp. 1541–1546.

[24] W. Enck, M. Ongtang, P. McDaniel, On Lightweight Mobile Phone Application Certification, in: ACM (Ed.), 16th ACM conference on Computer and Communications Security (CCS'09), 2009, pp. 235 – 254.

[25] M. Nauman, S. Khan, X. Zhang, Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints, in: ACM (Ed.), 5th ACM Symposium on Information Computer and Communication Security (ASIACCS'10), 2010, pp. 328–332.

[26] G. Dini, F. Martinelli, A. Saracino, D. Sgandurra, MADAM: A Multi-level Anomaly Detector for Android Malware, in: Proceedings of the 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security: Computer Network Security, MMM-ACNS'12, Springer-Verlag, 2012, pp. 240–253.

[27] A. Saracino, D. Sgandurra, G. Dini, F. Martinelli, Madam: Effective and efficient behavior-based android malware detection and prevention, IEEE Transactions on Dependable and Secure Computing PP (99) (2016) 1–1. `doi:10.1109/TDSC.2016.2536605`.

[28] D. Barrera, H. G. Kayacik, P. C. van Oorschot, A. Somayaji, A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android, in: ACM (Ed.), 17th ACM Conference on Computer and Communications Security (CCS'10), 2010, pp. 73–84.

[29] M. Backes, S. Gerling, C. Hammer, M. Maffei, P. Styp-Rekowsky, AppGuard – Enforcing User Requirements on Android Apps, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 7795 of LNCS, Springer Berlin Heidelberg, 2013, pp. 543–548.

[30] G. Dini, F. Martinelli, I. Matteucci, A. Saracino, D. Sgandurra, Introducing Probabilities in Contract-Based Approaches for Mobile Application Security, in: Data Privacy Management and Autonomous Spontaneous Security, LNCS, Springer Berlin Heidelberg, 2014, pp. 284–299.

[31] TrustGo, Trustgo website, Last access, 4th of Dec, 2015 (2014).
URL `http://www.trustgo.com/en/`

[32] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, I. Molloy, Using Probabilistic Generative Models for Ranking Risks of Android Apps, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12), ACM, 2012, pp. 241–252.

[33] Google Inc., Google verify app, Last access, 4th of Dec, 2015 (2014).
URL `https://goo.gl/pZgs5R`

[34] DAI-Labor, Androlyzer, Last access, 4th of Dec., 2015 (2014).
URL `https://www.androlyzer.com/`

[35] Webroot, Brightcloud mobile app reputation service, Last access, 4th of Dec., 2015 (2014).
URL `http://goo.gl/avXKXD`

[36] SAP AG, Securing Mobile Apps in a BYOD World, Last access, 4th of Dec., 2015 (2013).
URL `http://goo.gl/3Tk8pz`

[37] Samsung Electronics Co, White paper: Samsung knox premium, Last access, 4th of Dec., 2015 (2014).
URL `https://goo.gl/K5FBO3`

Appendix A. Excerpt of Analyzed Android Permissions

Table A.7: Partial list of Android Permissions and Associated Threat Levels, per Index

Permission	**Class**	**PT**	**ST**	**FT**
android.permission.ACCESS_COARSE_LOCATION	Dangerous	0.4	0	0
android.permission.ACCESS_FINE_LOCATION	Dangerous	0.8	0	0
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS	Normal	0.2	0	0
android.permission.ACCESS_MOCK_LOCATION	Normal	0	0.4	0
android.permission.ACCESS_NETWORK_STATE	Normal	0.2	0	0.4
android.permission.ACCESS_WIFI_STATE	Normal	0	0	0.4
android.permission.AUTHENTICATE_ACCOUNTS	Dangerous	0.6	0	0
android.permission.BATTERY_STATS	Normal	0	0.2	0
android.permission.BLUETOOTH	Dangerous	0.6	0.2	0
android.permission.BLUETOOTH_ADMIN	Dangerous	0.8	0.6	0
android.permission.BROADCAST_STICKY	Normal	0	0.2	0
android.permission.CALL_PHONE	Dangerous	0.6	0.2	1
android.permission.CAMERA	Dangerous	0.8	0.6	0
android.permission.CHANGE_CONFIGURATION	Dangerous	0	0.4	0
android.permission.CHANGE_NETWORK_STATE	Dangerous	0.2	0.6	0.6
android.permission.CHANGE_WIFI_MULTICAST_STATE	Dangerous	0	0.2	0.2
android.permission.CHANGE_WIFI_STATE	Dangerous	0	0.6	0.6
android.permission.CLEAR_APP_CACHE	Dangerous	0	0.2	0
android.permission.PROCESS_OUTGOING_CALLS	Dangerous	0.8	0.6	0.2
android.permission.READ_CALENDAR	Dangerous	0.8	0	0
android.permission.READ_CONTACTS	Dangerous	1	0	0
android.permission.READ_SMS	Dangerous	1	0	0
android.permission.RECEIVE_BOOT_COMPLETED	Normal	0.2	0.4	0
android.permission.RECEIVE_MMS	Dangerous	1	0	0.8
android.permission.RECEIVE_SMS	Dangerous	1	0	0.8
android.permission.RECEIVE_WAP_PUSH	Dangerous	0.4	0.6	0.6
android.permission.RECORD_AUDIO	Dangerous	0.8	0.6	0
android.permission.REORDER_TASKS	Dangerous	0.4	0.2	0.4
android.permission.RESTART_PACKAGES	Normal	0	0.2	0
android.permission.SEND_SMS	Dangerous	0.8	0.2	1
android.permission.WRITE_CALENDAR	Dangerous	0.8	0.2	0
android.permission.WRITE_CONTACTS	Dangerous	0.6	0.6	0
android.permission.WRITE_EXTERNAL_STORAGE	Dangerous	0.2	0.6	0
android.permission.WRITE_SMS	Dangerous	0.4	0.2	0
android.permission.WRITE_SOCIAL_STREAM	Dangerous	0.6	0	0
android.permission.WRITE_SYNC_SETTINGS	Dangerous	0	0.4	0