Evaluating the Trust of Android Applications through an Adaptive and Distributed Multi-Criteria Approach

Gianluca Dini*, Fabio Martinelli†, Ilaria Matteucci†, Marinella Petrocchi†, Andrea Saracino*†, Daniele Sgandurra†
* Dipartimento di Ingegneria dell'Informazione
Università di Pisa, Italy
name.surname@iet.unipi.it
† Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Pisa, Italy
name.surname@iit.cnr.it

***Abstract*—New generation mobile devices, and their app stores, lack a methodology to associate a level of trust to applications to faithfully represent their potential security risks. This problem is even more critical with newly published applications, for which either user reviews are missing or the number of downloads is still low. In this scenario, users may not fully estimate the risk associated with downloading apps found on online stores. In this paper, we propose a methodology for evaluating the trust level of an application through an adaptive, flexible, and dynamic framework. The evaluation of an application trust is performed using both static and dynamic parameters, which consider the application meta-data, its run-time behavior and the reports of users with respect to the software critical operations.**

We have validated the proposed approach by testing it on more than 180 real applications found both on official and unofficial markets by showing that it correctly categorizes applications as trusted or untrusted in 94% of the cases and it is resilient to poisoning attacks.

I. Introduction

App stores for mobile devices are continuously growing both in their number and in the amount of proposed applications. Android is currently the mobile platform with the largest market share and its official market, *Google Play*, hosts hundred of thousands of applications that are continuously published by third-party developers. Users browse app-stores to choose and install applications on their smartphones or tablets, either for free or paying the app's price established by the developer. However, both official and the unofficial markets do not perform strict controls on published applications. In fact, even on official markets, applications infected with malicious software (malware) have been found [1].

Malicious apps may leak user credit, private data, and even cause physical damage to the device itself. The native Android security mechanisms have proven to be not so effective in protecting users and devices from some kind of malware, such as trojanized apps. The main reason is the semantic of the *Android Permission System*, which requires a user to be able to understand the hazardousness of an application by reading at install-time the list of resources the application should access. Several critics have been raised against this system, since a large share of Android users does not fully understand the permissions required by an application [2].

In this paper, we propose *MAETROID* (Mobile Application Evaluator of TRust for andrOID), a framework for evaluating the level of trust of Android applications, which extends the framework proposed in [3]. This framework estimates the level of trust for an application using static analysis only. However, since several applications lack some meta-data used to estimate their level of trust, MAETROID collects information about the *dynamic behavior* of applications and provides feedback concerning software issues, such as bugs, which may be symptoms of security threats. These pieces of information are collected by a central server and become available to other users that may download and install the same application. In this way, MAETROID evaluates the trust level for an applications on the base of both a distributed user-experience system and a set of static meta-data. MAETROID has proven to be effective in helping users to classify the level of trust of applications, especially for brand-new applications (from official and unofficial markets) and those coming from markets that do not provide rating scores.

The remainder of the paper is organized as follows. Section II describes some related works. Section III briefly recalls the framework proposed in [3], which is extended by the work proposed in this paper. Section IV describes MAETROID, the voting architecture and the score system to evaluate the trust level of applications. Some results and the resilience to some attacks aimed at forging the feedback system are discussed in Section V. Finally, Section VI briefly concludes by proposing some future extensions.

II. Related Work

This work extends [3], which proposes a multi-criteria evaluation of Android applications to help the user to easily understand the trust degree of an application, both from a security and a functional point of view. Several extensions and improvements to the Android permissions system have been recently proposed. [4] proposes a security framework that regulates the actions of Android applications defining security rules concerning permissions and sequence of operations. New rules can be added using a specification language. The application code is analyzed at deployment-time to verify whether it is compliant to the set of rules, if not it is considered as malicious code. Our proposal does not require the code to be decompiled and analyzed, since it only requires the

permissions list that can be retrieved from the manifest file and other generic information that can be retrieved from the website where the application can be downloaded.

Authors of [5] 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 permissions she wants to grant to an application and those that have to be denied. Using data mocking, they ensure that an application 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 [6]. The authors designed an improved application installer that allows 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.

In [7], applications have been classified based upon their required permissions. Applications have been divided in functional clusters by means of Self Organizing Maps, by proving that apps with the same set of permissions have similar functionalities. However this work does not differentiate between good and trojanized apps. Finally, another analysis of Android permissions is presented in [8], where the authors discuss a tool, *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 return a mapping between functions and permissions. [9] discusses whether users can be trusted to assess the security of an app. In particular, the authors explore the security awareness of users when downloading apps from official markets. The paper shows that most of the users trusts the repository regardless of the security alerts during application selection and installation. Hence, there is a need of a simple mechanism that helps users to easily and clearly understand the security criticalities of an application. We propose MAETROID as a possible solution.

In [10], a set of comments on Android Market has been manually analyzed to verify if users are concerned about security and how much they speak about security and permissions in the comments. Only 1% of the comments contains references to application permissions. Hence, a framework that analyzes application permissions, such as MAETROID, is needed. The security issues brought by unofficial application markets have been also addressed in [11], which presents a framework that leverages application meta-data to detect malicious applications coming from unofficial markets. The framework also implements a kill-switch to uninstall detected malicious applications from all of the smartphone that install it, extending this feature of the official market to unofficial ones.

III. Assessment of Android Apps through AHP

In this section, we recall the approach for assessing the trust level of an Android application proposed in [3]. The approach exploits the AHP decision process briefly described hereafter.

The Analytic Hierarchy Process (AHP) [12] [13] is a multi-criteria decision making technique, which has been largely used in several fields of study. 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 previously established *criteria*. The problem is structured as a hierarchy, as shown in Figure 1, by linking goals and alternatives through the chosen criteria. This approach requires to subdivide a complex problem into a set of sub-problems, equal in the number to the chosen criteria, and then to compute the solution by properly merging the various local solutions for each sub-problem. Local solutions are computed by means of *comparison matrices*, that describe for each criterion how much an alternative is more relevant with respect to another one in a pairwise fashion.

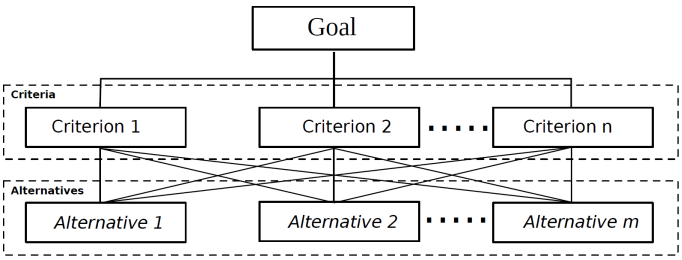

Figure 1: Generic AHP Hierarchy

In [3] the authors describe an instantiation of the AHP decision methodology to asses the trust level of an Android application as follows. Given an Android application with the following parameters:

- a threat score σ,
- a developer δ,
- a number of downloads η,
- a market μ,
- a user-rating ρ,

then the *goal* consists in assigning to an application one of the following *alternative* labels:

- *Trusted*: the application correctly works and should not hide malicious functionalities;
- *Untrusted*: the application could violate the security of the mobile device;
- *Deceptive*: the application is neither functional nor secure.

The problem is parametric with respect to the application and, for two different applications, the same fixed alternatives have a different relevance to the same criterion. Hence, the five parameters (σ, δ, η, μ, ρ), which are the *criteria* of the problem, assume different values for different applications. To each value of the five criteria is assigned a different comparison matrix, leading AHP to take a different decision. Notice that AHP asseses the decision using the parameters all together. Each parameter taken as stand-alone may be not significant, but is however helpful to the decision process.

We have developed a framework to analyze and classify Android applications, which fully implements the previous strategy [3]. The test-set was composed of 180 Android applications of different categories, which are known to be:

- *Good App*: an application that behaves correctly both from the security and functional point of view. These applications may come from official markets or unofficial ones: in the last case, sometimes AHP may not have enough data (e.g., user rating and number of downloads) to asses the quality of the applications and, hence, the application is considered untrusted. However, the adaptive solution that we propose in this paper may supply this lack of information as soon as a sufficient number of user reports have been issued. An application in this category should be classified as *Trusted*;
- *Infected App*: an application infected by a malware. An application in this category should be classified as *Untrusted*;
- *Bad App*: an application that is malfunctioning (e.g., it often crashes) or unusable, and it is not trojanized. An application in this category should be classified as *Deceptive*.

Notice that the three classes are not overlapping, i.e. they are composed of three disjoint sets of applications. The framework proposed in [3] assigns to all these applications one of the previous alternative label by instantiating the AHP decision procedure. Hence, the goal is to correctly assign labels to applications, i.e. *Trusted* to *Good App*, *Untrusted* to *Infected App* and *Deceptive* to *Bad App*. However, in some cases, since some applications lack some meta-data used to properly estimate their level of trust, the framework [3] could classify them as *Untrusted*, even if these application were *Good App*. In the next Section, we discuss an improvement to the framework to address this issue.

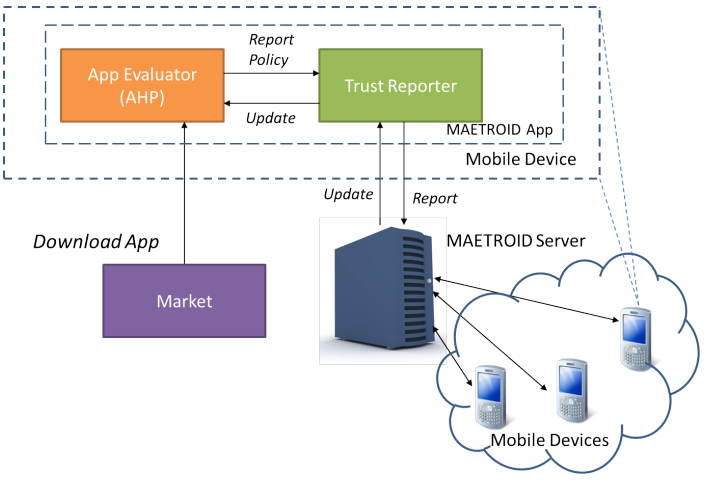

Figure 2: Framework Architecture

IV. Adaptive Framework

The framework described in the previous section computes a first evaluation of an application level of trust using static information only. However, some unofficial markets do not provide user ratings or the number of downloads and, hence, some applications coming from these markets can be considered as *Untrusted* by the decision system even if they are *Good Apps* or vice-versa.

A. Architecture

To address this problem, we propose *MAETROID* (Mobile Application Evaluator of TRust for andrOID), a dynamic and distributed user-based evaluation protocol, which allows smartphones to periodically exchange feedback on applications' behavior. The MAETROID framework is depicted in Fig. 2, where the application (MAETROID-App) implements both AHP and the new adaptive protocol. Basically, once a user has installed MAETROID-App, an evaluation module implements the standard AHP algorithm on a newly-downloaded application (Sect. III) and produces a first decision on the application trust level. Then, MAETROID-App allows users to collaborate to a reporting network to produce feedback on the app's (mis-)behavior. These reports are automatically sent by the MAETROID-App to the MAETROID server (see Fig. 2), which analyzes the received data to build AHP matrices for a new criterion, which we call *user-experience*, for the AHP decision process.

The rationale behind this new criterion is that, if several users notice that an application contains several bugs, or it continuously crashes, or its usage strongly affects the battery duration, then the AHP decision should be recomputed and either shifted towards the *Deceptive* or *Untrusted* alternative, since these features may be symptom of an application hiding malware. Conversely, a *Good App* from unofficial markets (or one that initially has a low score), which may be initially categorized in the *Untrusted* class, after receiving several good reviews from users, should be labelled as *Trusted*.

To evaluate if an application misbehaves, users send feedback through a report[1], which can be easily understood and filled in (directly on the MAETROID-App) by an average user. The report is based upon the following five parameters, which we have chosen as indicative of the application behavior and that may be signals of several misbehaviors:

- crashes (CR): if the application has ever terminated unexpectedly;
- battery depletion (BD): the amount of battery consumed as experienced by the user;
- usability (US): evaluation of the application responsiveness;
- credit leakage (CL): if the application deceitfully consumes user credits;
- misbehavior (MI): if the application shows some critical bugs, in a way clearly observable by the user.

For example, the battery depletion may be caused by a malicious and hidden use of smartphone sensors by a malware. For each parameter, users select one out of three possible values, which are listed in Table I.

[1]Notice that the report differs from the app rating, which considers the app functionalities, while a report only considers app misbehaviors.

Parameter	Possible Values		
CR	Never	Seldom	Often
BD	Avg	Slight	Strong
US	OK	Some Issues	Unusable
CL	None	Maybe	For Sure
MI	None	Few	Several

Table I: Report Parameter

Value of S	Issues
7	No issues
6	Negligible issues
5	Lesser issues, such as faster battery depletion
4	Neutral, non evaluated or some issues
3	Several Issues and bugs
2	Major bugs or strong battery depletion
1	Critical security or functional issues

Table II: Score Interpretation

Each value in Table I is associated with a number in the range $[0, 6] \in \mathbb{N}$. We denote as r_i the score of the i-th feedback received by the MAETROID-server for an app, defined as:

$$r_i = 7 - (CR + BD + US + CL + MI)$$

If r_i is negative, the server changes its value to 1, hence $r_i \in [1, 7]$, where a low value means a bad behavior and a high value means a good behavior of the app. For each app a, for which at least one report has been received, the MAETROID-server stores a score $s_i(a)$ (henceforth s_i) as defined in Eq. 1.

$$s_i = (1 - \alpha)s_{i-1} + \alpha r_i, \quad s_i \in [1, 7] \tag{1}$$

In Eq. 1, which is the canonical form of a convex hull (with $\alpha \ll 1$), the new value of s_i is computed in a way where the last received report r_i slightly affects the new value of s_i.

We assess the value of $\alpha : \mathbb{R}^+ \to [0, 1]$ as a function of the number of the expected received reports i for an application. To estimate the value of α we suppose that the MAETROID-server receives for the same application an amount i of costant valued reports, to change the value stored from an initial value s_0 to a final chosen value s_i. Thus, solving the following equation we compute the value of α:

$$\alpha(i) = \frac{\gamma}{i}, \quad \gamma = s_i - s_0$$

For example, let us consider an application with an initial neutral score $s_0 = 4$ (that is exactly what happens when a new application is added to the server). We want to compute the correct value of α to shift the score of the app to the value of $s_i = 7$, after receiving 1000 reports of value $r_i = 7$. The resulting value of α is $\alpha = 3 * 10^{-3}$. The number of expected received reports is parametric with the number of users that have downloaded the application under evaluation.

B. Re-Assessing the Decision

To reassess the decision, the server constantly updates a value $S \in [1, 7] \subset \mathbb{N}$, which is computed by rounding the current value of s_i. S is used as an index to choose the comparison matrix that will be used by the app evaluator (AHP) for the new criterion, called *user experience*.

For the *user experience* criterion we define seven AHP comparison matrices $\{U^1, \ldots, U^7\}$, one for each possible value of S, where U^4 is a neutral comparison matrix, i.e. each matrix element $U^4_{i,j} = 1, \forall i, j$. This matrix is assigned to an application that has never received reports. On the other hand, U^1 is a matrix assigned to an application that has received very bad reports, and U^7 is assigned to an application that has received several good reports (see Fig. 3 and Tab. II).

When evaluating a particular application, the newly proposed AHP instantiation considers the criteria listed in Section III, plus the criterion *user experience*. The matrix U^S (where S is the current score for that application) is added to the whole AHP decision process. This matrix compares the three alternatives, *Trusted*, *Untrusted*, and *Deceptive* with respect to the value of S.

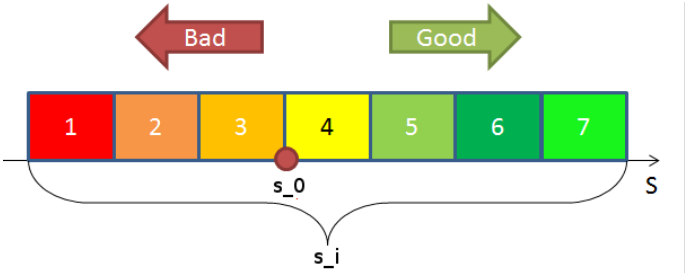

Figure 3: Values of Index S

After each received report, the server computes s_i as in Eq. 1, rounds its value and if a change in S occurs then it triggers again the AHP decision process. In this way, the new comparison matrix U^S is used by AHP decision process, and the app's level of trust is recomputed. Hence, new users installing the considered application will receive the updated decision, whereas users that already own this app will be notified by MAETROID-App only if the new value of S causes a change in the decision process, e.g., a *Trusted* application becomes *Untrusted* (due to several bad reports). However, the score s_i on the MAETROID server is updated each time a user sends a new report.

A common issue of user report-based systems is the possibility for an attacker to send fake reports, for maliciously poisoning the app's evaluation. Common patterns of these attacks consist of sending burst, i.e., a large amount of good (or bad) reports sent in a small amount of time by groups of users that collude to maliciously increase (or decrease) the score of an app. To address this issue, we propose a modified version of Eq.1 that considers the time elapsed between two reports:

$$s_i = (1 - \beta)s_{i-1} + \beta r_i, \quad \beta = \alpha\left(1 - \frac{1}{\Delta t}\right) \tag{2}$$

With Eq. 2, a burst of reports received in a small time interval

slightly influences the value of s_i; in fact, the value of β ranges in the interval $[0, \alpha]$ and it is a function of Δt. Δt is the time elapsed between two successive reports t_i, t_{i-1} and it is defined as:

$$\Delta t = \begin{cases} 1 & \text{if} \quad t_i - t_{i-1} < 1 \\ t_i - t_{i-1} & \text{otherwise} \end{cases}$$

Hence, the value of β, that is the weight for the last received score, is closer to zero for small values of Δt, so that a burst of votes will slightly affect the score s_i of the application (and consequently S). Notice that is unlikely that several non-colluding users send a report for the same application at the same time. Hence, the influence of the Eq. 2 on the effectiveness of genuine reports should be neglectible.

V. Results

To prove the effectiveness of our approach, we have tested MAETROID on the same set of 180 applications used in [3]. We remind the reader that this acts as a validation set, since we know in advance which one is a *Good*, *Bad*, or *Infected* app (see Section III). Furthermore we have run several simulations taking into account also user reports. In particular, we have tested:

- the effectiveness of changing the decision from *Untrusted* to *Trusted* for a brand new application that receives positive reports;
- the resilience to reputation tampering attacks;
- the resilience to reputation tampering attacks with report bursts.

To this end, we have built a test-bed to simulate the reports using the following series:

1) a series of i reports, all with the maximum score and constant inter-arrival time;
2) a series of i reports, all with the minimum score and constant inter-arrival time;
3) a series of i positive reports, with scores spanning from five to seven. The inter-arrival times of the reports have been chosen according to a Poisson process of parameter λ;
4) a series of i negative reports, with scores spanning from one to three. Poisson process inter-arrival time.

The parameter λ used in series 3) and 4) depends on the average number of reports received in a time interval for a specific application. The process of users downloading an app and sending reports, starting from the time when an application becomes available on a market, can be modelled as a Poisson process. The series 3) and 4) represent possible sequences of votes. Notice that series 1) and 2) are not realistic report sets. These two series have been used as extreme test case to raise or lower the score of an application, or to simulate vote bursts.

In the first experiment, we have considered all the applications in [3] that have been classified as *Untrusted* due to the lack of users' ratings. As anticipated before, actually those applications should have been classified as *Trusted*. As a sample, we report the classification process on `Candy Zuma`, a simple video-game found on an unofficial market that does not provide user ratings. Candy Zuma is a good app, however, in [3], it has been labelled as *Untrusted*. The application has been downloaded by more than 1000 users. Using both the series 1) and 3), with $i = 100$, during the simulations the application receives a score S equals to 7 and 6, respectively. Using either U^7 or U^6 as comparison matrix for the newly introduced criterion *user experience*, the decision is changed from *Untrusted* to *Trusted*.

Fig. 4 shows a comparison between prior classifications, obtained without the *user experience* criterion, and current classifications, with the use of the new criterion. We can notice that most of those apps coming from unofficial markets ("*Good-Apps* (Unoff)" bar), which have been labelled as untrusted, are now correctly classified.

Globally, all the infected apps of our validation set have been correctly classified by MAETROID, which recognizes them as untrusted apps. The same holds for bad apps: all of them have been classified as deceptive. The 91% of good apps have been classified as trusted. Thus, the new framework achieves a global accuracy of 94%. All of the 24 applications are considered *Trusted* using the matrix U^7 in the AHP decision process. Using U^6, AHP outputs the same decision on 13 applications out of the 24 wrongly considered *Untrusted* in [3].

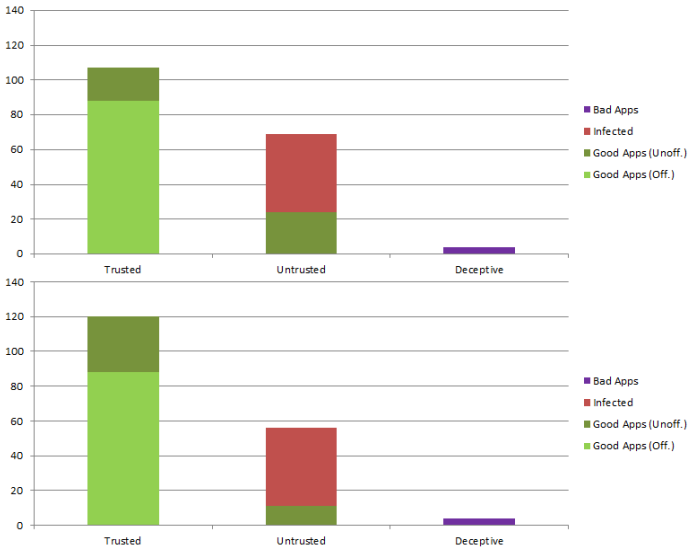

Figure 4: Classification Without (Top) and With (Bottom) The User-Experience Module

Eq. 1 avoids strong fluctuations of an application score s_i. In fact, the score changes only after the reception of a sufficiently large number of users' reports, with the same trend (i.e., all positive or negative scores). If we consider popular applications, such as the well-known game `Angry Birds`, downloaded from the official market more than 100 millions of times and rated by more than 1 million of users, we can see that this kind of app is characterized by a very small value of α. In fact, if we suppose $i = 100K$, then we have $\alpha = 3{\cdot}10^{-5}$. Hence, an attacker that wishes to tamper the user's score of `Angry Birds`, should be able to send a very large number of negative reports, meaning that she needs, at least, to cooperate with a very large community of colluding users. Moreover, the

reports should be well distributed in time and not interlaced with other positive reports.

To show the robustness of our approach, we have raised the score of the application to 7 using the series 1) with $i = 1K$ and the series 3) with $i = 100K$. Then, we have interlaced a series 3) to simulate others incoming good rates, and a series 4) to simulate the reputation tampering attack, both with $i = 100K$. Such a strong attack has lowered the application score of only one point. The AHP decision for `Angry Birds` remains *Trusted* also with U^6 in place of U^7 and, hence, the attack is ineffective.

The parameter β allows the score system to be protected by bursts of reports that can be sent, for example, from a botnet. We have tested the effect of a burst of $i = 100K$ negative reports with an inter-arrival time of 1 sec on `Angry Birds`. The burst of reports does not affect the score of the application.

The same reasoning applies to the opposite situation, in which a burst of positive reports is sent to artificially promote a new application. Considering a new application with a neutral score of 4, the score increases only if a sufficient number of votes is received, which are fairly distributed in time and that they have the same positive trend. Otherwise, the score is not modified and the AHP decision is not affected.

VI. Conclusions and Future Work

In this paper, we have presented *MAETROID*, a framework for assessing the level of trust of Android applications by analyzing both static meta-data and user feedback collected in a distributed manner. MAETROID firstly analyses the apps meta-data and outputs an initial decision about the application level of trust. Then, static criteria are merged with a score index that dynamically change. The reputation index is used as a new decision criterion, called *user-experience*, and it is based upon reports sent by users to a centralized server. The server interacts with MAETROID each time a new application is installed, by providing updated information to assess the application behavior and its level of trust. The effectiveness of the proposed framework has been tested on a set of 180 applications constituting our validation set, and it has been proven to be effective, in particular, in classifying the level of trust of brand-new applications and other apps whose markets does not provide rating indexes. Finally, MAETROID is resilient to reputation tampering attacks and vote bursts.

As future work, we plan to distribute the applications to a large number of users, to verify the compliance of the proposed model with real use-cases. It is worth noticing that the reporting architecture can also be built on a pure P2P network or a hybrid one, where each node (reviewer) can have a different level of reputation and the decision can be weighted according to this value. In fact, in the current model we consider all the reviewers as equally trusted. Finally, we are going to extend MAETROID to automatically assess the value of the five parameters used to build the report from which the user-experience score is derived.

References

[1] Xuxian Jiang, "Multiple Security Alerts: New Android Malware Found in Official and Alternative Android Markets," 2011, http://www.csc.ncsu.edu/faculty/jiang/pubs/index.html.

[2] A.P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, D. Wagner, "Android permissions: User attention, comprehension, and behavior," Electrical Engineering and Computer SciencesUniversity of California at Berkeley, Tech. Rep., 2012, http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-26.html.

[3] G. Dini, F. Martinelli, I. Matteucci, M. Petrocchi, A. Saracino, D. Sgandurra, "A Multi-Criteria Based Evaluation of Android Applications," in *4th International Conference on Trusted Systems, InTrust 2012*. Springer-Verlag, December 2012.

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

[5] 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)*, June 2011.

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

[7] 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 *17th ACM Conference on Computer and Communications Security (CCS'10*. ACM, October 2010.

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

[9] A. Mylonas, A. Kastania, and D. Gritzalis, "Delegate the smartphone user? security awareness in smartphone platforms," *Computers & Security*, vol. 34, no. 0, pp. 47 – 66, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404812001733

[10] E. Ha and D. Wagner, "Do android users write about electric sheep? examining consumer reviews in google play," in *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, 2013, pp. 149–157.

[11] D. Barrera, W. Enck, and P. van Oorschot, "Meteor : Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems," in *Mobile Security Technologies Workshop (MoST)*. IEEE, 2012. [Online]. Available: http://www.ccsl.carleton.ca/ dbarrera/files/most12-barrera.pdf

[12] T. L. Saaty, "Decision-making with the ahp: Why is the principal eigenvector necessary," *European Journal of Operational Research*, vol. 145, no. 1, pp. 85–91, 2003.

[13] ——, "Decision making with the analytic hierarchy process," *International Journal of Services Sciences*, vol. 1, no. 1, 2008.