A Distributed Framework for Collaborative and Dynamic Analysis of Android Malware

Mario Faiella, Antonio La Marra, Fabio Martinelli, Francesco Mercaldo, Andrea Saracino, Mina Sheikhalishahi
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Pisa, Italy
Email: name.surname@iit.cnr.it

Abstract—Combination of dynamic and static analysis is very effective in detecting malicious Android apps. However, dynamic analysis is hardly practiced on large scale, due to the necessary active interaction with the malicious app, which is reliable only if performed by a user on a real device. In this paper we present a framework for distributed and collaborative analysis of Android suspicious apps, which leverages real users to test the functionality of apps and detect eventual malicious behaviors by exploiting an on-host app for intrusion detection. The paper introduces the architecture, workflow and protocols to handle the report received by participating users, detecting and filtering the malicious ones. Simulative results to assess the performance of the proposed framework are reported and discussed.

I. INTRODUCTION

Android malicious applications (apps) are raising exponentially in number and complexity [1]. Over one million of malicious apps is currently distributed in the wild, showing always new attack vectors, together with several methodologies to hide the malicious code and cloaking its effect, deceiving both users and antivirus software [2]. Early detection of Android threats requires an active monitoring of the main channels used for app distribution, and rigorous app analysis which should not be limited to the standard, easy-to-deceive signature matching. In fact, to maximize the possibility of detecting threats, such as zero-day attacks, the detection methodology should join static and dynamic analysis techniques [3], mainly based on behavioral analysis, hardly deceived by common obfuscation techniques [4]. However, this analysis process is computationally heavy and time consuming, hence difficult to apply on a large scale as a standalone service. This consideration is further worsened by the fact that often dynamic still automatic analysis is not effective, due to the ability of some malware samples able to recognize when they are executed inside an automated testing environment such as sandbox and emulators [5]. Hence, app analysis might require direct interaction with real users, which maximize the needed effort and can make the malware analysis for early detection of new threats, unfeasible on a large scale. We argue that for a possible solution, it is necessary to design a collaborative approach, which leverages on real smartphone users, with a modest security awareness, to analyze several malware samples in a shorter amount of time, still able to handle the introduced issues on trust and reliability.

In this paper we propose D-BRIDEMAID (Distributed BRIDEMAID) a distributed framework for cooperative analysis of Android apps, aimed at early detection of new threats through distributed dynamic analysis. The framework leverages on BRIDEMAID (Behavior-based Rapid Identifier Detector and Eliminator of Malware for AndroID), an accurate software for malware detection based on static and dynamic analysis, which obtains an accuracy of 99.7% on a dataset of almost 3k real world malicious apps. This extension of the framework proposed in [3], is designed to automatically analyze the behavior of apps on the devices of a pool of users, collecting their reports and deciding on the trustworthiness of the tested app. These users, willing to accept to participate to a collaborative program for malware detection in which they install new apps, potentially malicious, on their device where the analysis software is running. Based on the observed behaviors and eventually raised alerts, the user can send a report for every tested apps, deeming it as Genuine or Malicious. The report of different users for the same app are collected to assess whether an app is malicious or not, and weighted according to parameters concerning the user reputation and the reliability of report, based on analysis time and additional parameters which will be discussed in the following. Since the envisioned model is completely Peer-to-Peer (P2P), without hypothesis of trust on users, the proposed framework exploits a reputation management algorithm to detect and tackle eventual attackers, not considering their reports, once they have been detected.

The contribution of the paper are in the following:

- We propose a distributed framework for analysis of Android malicious apps which exploits a set of cooperating users and advanced techniques for monitoring and detect malicious behaviors based on static and dynamic analysis.
- A mathematical model is proposed to evaluate the reliability of different reports and combining them into a single index which expresses the overall reliability of an app evaluation.
- We report a set of features relevant in evaluating the reliability of an app testing, also related with specific behaviors.
- A set of simulative experiments is reported to evaluate the proposed framework and mathematical model, measuring its reliability at the variation of the number of attackers.
The remainder of the paper is organized as follows. Section II recalls the background concept of static, Meta data, and dynamic analyses of Android apps. Section III describes the proposed framework, attacker model, and the workflow. Also it introduces the theoretical aspects of reliability, users’ reputation and result’s validity calculation, and Section IV validates our framework showing a set of experimental results. Finally, Section V reports some related work, while Section VI concludes the paper.

II. BACKGROUND

In this section we describe the BRIDEMAID approach, the on device analysis framework for Android apps which combines static and dynamic techniques to discriminate Android malware applications from legitimate ones. The proposed framework leverages on a multi-level and multi-feature analysis which includes permission scoring and evaluation, opcode analysis, kernel level monitoring and API calls hijacking. Further details can be found in [3].

A. Static Analysis

Relating to static analysis, we consider a binary classification problem in which an unknown application has to be classified as malicious or genuine. The static analysis phase consists of two phases: a learning phase, in which the classifier is trained using a labelled dataset of applications, and the classification phase, in which an input application is classified as malicious or genuine. We consider as feature n-grams of smali code with n = 2 because a previous work [6] demonstrated that the sequences of two consecutive opcodes obtain better performance in Android malware identification. Furthermore, BRIDEMAID performs an analysis of meta-data extracted from the manifest file to analyze the required permissions and compute a threat score. In addition we consider the market of provenance, the download number, the user rating and the developer reputation (if available). All these parameters are combined through the Analytic Hierarchy Process [7], which decides if the new app should be considered Trusted or Suspicious. Apps which are considered trusted can be executed on the device without additional check on the performed behaviors. Apps which instead are classified as suspicious will be subject to the control of the dynamic analysis monitors, as discussed in the following.

B. Dynamic Analysis

The dynamic analysis, considers both global features, i.e. related to the device and operative system, and local features which are related to specifically monitored apps. The dynamic analysis is based on two core elements that monitor different sets of features: the Global Monitor and the Per-App Monitor. The Global Monitor monitors the device and OS features at three levels, i.e. kernel (SysCall Monitor), user (User Activity Monitor) and application (Message Monitor). These features are monitored regardless of the specific app or system components generating them, and are used to shape the current behavior of the device itself. Then, these behaviors are classified as genuine or malicious by a classifier. The Per-App Monitor, implements a set of known behavioral patterns to monitor the actions performed by the set of suspicious apps.

III. ARCHITECTURE AND WORKFLOW

In this section we will describe the architecture of D-BRIDEMAID, introducing the components, their interactions and the operative workflow.

A. System Model

Figure 1 depicts the main components of the envisioned architecture. As shown, the apps are stored in a cloud storage which also acts as orchestrator for the whole framework. The main actors of the framework are the app testers, which are users that agree to participate to this app evaluation process. Tester devices are all equipped with D-BRIDEMAID, which acts as a dynamic Intrusion Detection System (IDS), reporting to the user suspicious behaviors. The testers willing offer their devices and their app interaction time, being aware that both their mobile devices and the contained information might be exposed to the risk of malicious apps. This risk is strongly mitigated by the effectiveness of BRIDEMAID, which detects and stops malicious identified behaviors. Moreover, testers receive a reward for the service they are offering and the risk they are taking. The reward, which could also be monetary-based, is used as incentive for user participation and together with a reputation-based algorithm discussed in the following, fosters a correct user behavior. After evaluating an app, the user submits a report, based on eventual BRIDEMAID alerts, deeming the tested app either as malicious or genuine. Reports of testers concerning every analyzed app are then collected by an Aggregator which gets a decision on the app trustworthiness, out of all the received reports.

B. On Device App

The D-BRIDEMAID framework is composed by an host application to install on the tester device. It is a lightweight application and does not hinder the everyday smartphone usage.
Once installed, the D-BRIDEMAID host application authenticates the tester through the IMEI and the IMSI in order to ensure that the application to test is running using a real environment and not on an emulated one (we check this because mobile malware usually do not perform malicious action if executed on emulated enviroment, this happens usually to elude honeypot). The D-BRIDEMAID application, once the user is authenticate to the framework, proposes a list of unknown applications to test. The D-BRIDEMAID application shows, for each application, the results derived from static analysis module, and the tester will decide whether run the application. In this case D-BRIDEMAID will install and launch the application to test on the device. Once the application to test is running, D-BRIDEMAID is able to check the background or foreground time of application, these parameters are useful to compute the user reliability, as explained in the next section.

D-BRIDEMAID host application is also capable to intercept a set of legitimate events that typically stimulate a malicious payload. An event in Android is sent from an application (i.e., the gps localization, the presence of WiFi newtorks) and from the device user (i.e., boot the device, charge the phone).

Table I shows the most occurring events able to activate the malicious behavior, which the percentage that the considered event is able to trigger the malicious payload. As matter of fact, there are several ways that a malware may employ to be activated, each one associated with an activating system event. This list and the percentage of malicious payload activation has been compiled taking into account the events which most frequently trigger the payload in Android malware, according to several studies [8].

<table>
<thead>
<tr>
<th>#</th>
<th>Event</th>
<th>Description</th>
<th>App Behavior</th>
<th>User Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BOOT</td>
<td>boot completed</td>
<td>80%</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>CALL</td>
<td>incoming call</td>
<td>30%</td>
<td>75%</td>
</tr>
<tr>
<td>3</td>
<td>SYS</td>
<td>phone rooted</td>
<td>30%</td>
<td>35%</td>
</tr>
<tr>
<td>4</td>
<td>BATT</td>
<td>battery status change</td>
<td>50%</td>
<td>85%</td>
</tr>
<tr>
<td>5</td>
<td>SMS</td>
<td>reception of SMS</td>
<td>70%</td>
<td>85%</td>
</tr>
<tr>
<td>6</td>
<td>NET</td>
<td>connectivity change</td>
<td>50%</td>
<td>80%</td>
</tr>
</tbody>
</table>

In Table I, the first row represents the **BOOT** event, which is the most used within existing Android malware. This event will be triggered and sent to all the applications installed in an Android device as the system finishes its booting process, a perfect timing for a malware to kick off its background services. By listening to this event, the malware can start itself without user’s interventions or interactions with the system. Another event used from malware writers is the **CALL** one (second row in Table I) event: this event will be sent in broadcast to the whole system (and all the running applications) when a new CALL is being received. The **SYS** events is accepted by rooted device, as matter of fact due to their open-source nature, many users are able to root their device in order to customize it and to expand their functionality. The **BATT** event is triggered when the power is connected or disconnected and when the operating system send the battery low and battery ok signals. The **SMS** event is transmitted to the system when a new SMS message is received. With this event, the malware has the ability to respond to specific incoming SMS messages to undertake malicious actions. The last event, the **NET** one is transmitted when a change in the data connection happens, for instance when the connection switches from GPRS to HSDPA network.

### C. Workflow

Being a distributed system, D-BRIDEMAID needs to define a workflow to synchronize the actions of all users testing apps. This workflow must be designed in the direction of flexibility, to ensure the highest possible participation for each analysis, which might also include the possibility of having the same user testing at the same time two different apps. Duration of app testing should be long enough to maximize the probability that eventual malicious code activates, still limited to allow a timely decision on the app trustworthiness.

We assume the system to own at each time a set of \( n \) active users (testers), i.e. currently able to install and test the functionality of apps. Apps to be tested are chosen by D-BRIDEMAID from a repository of apps considered suspicious after a first static analysis done by the system exploiting opcode n-grams and app metadata. D-BRIDEMAID chooses a short list of apps, proposing them to the \( n \) users. D-BRIDEMAID can decide which apps to propose to each user, in order to balance the number of users testing each app. Each user can choose among the proposed ones, the apps to test. Once a user accepts, the app is downloaded and installed on the device and the test can be run for a configurable time span named **evaluation round**, of duration \( t_r \). From the time the first user installs the app, any other invited user can start its evaluation round, given that accepts the invitation before the evaluation round of the first user ends. The duration of the evaluation round is the same for each user, whilst the whole duration of an app evaluation is named **evaluation epoch**, having a variable duration named \( t_e \). Due to the constraints on the joining time, it is ensured that the duration of an evaluation epoch cannot be longer than twice the evaluation round, i.e. \( t_e \leq 2t_r \). This workflow is depicted in Figure 2, showing an example in which four testers join to an evaluation epoch. The event of joining is represented by the small arrows. As shown, after the end of the first evaluation round, no other testers can join to the evaluation. At the end of her own evaluation round, the user produces a **report**. The report contains the following information: a binary decision to classify the app as malicious or genuine. At the end of the evaluation epoch,
the aggregator collects all the reports concerning the analyzed app and computes a decision based on the aggregated results. After the decision has been taken, tester reputation is updated according to Algorithm 1 and revenue is assigned to users providing a useful, i.e. reliable and correct result.

As additional notes, the service is able to handle parallel session, with users testing on the same device different apps at the same time. However, for each app there is a specific evaluation epoch, not necessarily synchronized with the evaluation epochs of the other apps.

### D. Reliability Computation

This section defines the reliability of a tester report as a function, which returns a real number in [0, 1]. To shape the reliability function, the following components are considered:

- **BRIDEMAID report:** if a user determines that an application α is malicious and she declares that her evaluation result is based on BRIDEMAID report on her device, then the provided score is given if the app has been in background or foreground for a time as close as possible to τi.

- **Time:** if the app is considered genuine, an higher reliability score is given if the app has been in background or foreground for a time as close as possible to τi.

- **Events:** additional reliability is assigned to report if one or more of the events in Table I is performed. In the following, we detail the aforementioned components and their impacts on reliability score.

Let user u report that the application α is malicious, at time t. We denote it as \( f_i(\alpha, t) = 1 \), where \( f_i \) is a Boolean function which returns 0 if α is genuine or 1 if malicious. Formally:

\[
 f_i(\alpha, t) = \begin{cases} 
 1 & \text{if } u_i \text{ reports } \alpha \text{ is malicious at } t \\
 0 & \text{if } u_i \text{ reports } \alpha \text{ is genuine at } t
\end{cases}
\]

**a) BRIDEMAID Report:** If user \( u_i \) reports the application \( \alpha \) as malicious, then the maximum reliability score should be returned. Formally, we define:

\[
 f_i(\alpha, t, \mathcal{B}) = \begin{cases} 
 1 & \text{if } u_i \text{ reports } \alpha \text{ is malicious at } t \\
 0 & \text{otherwise}
\end{cases}
\]

**b) Time:** Let \( t_0 \) be the beginning of the first evaluation round for \( \alpha \). We denote by \( t \) the amount of time passed from \( t_0 \); by \( T \) the maximum time duration, i.e. equivalent to evaluation epoch; and \( t_i(\alpha, t) \) the duration of time that application \( \alpha \) has been active in background or foreground for user \( u_i \). Hence, \( t_i(\alpha, t) \in [0, T] \).

Moreover, the time should affect the reliability in exponential ascending scheme, since the probability of an app activating the malicious payload is high at the beginning and exponentially decreases during time. Therefore, we define the following function for computing the impact of time in reliability of the report of user \( u_i \) about app \( \alpha \):

\[
 \tau_i(\alpha, t) = 1 - e^{-t_i(\alpha, t)} \quad t_i(\alpha, t) \in [0, T]
\]

**c) Events:** Representing by \( \mathcal{B}(t), \mathcal{C}(t), \mathcal{Y}(t), \mathcal{M}(t), \mathcal{A}(t), \mathcal{S}(t) \) and \( \mathcal{N}(t) \) the Boolean functions of events reported in Table I respectively, reported by user \( u_i \) at time \( t \), such that it equals to 1 if user \( u_i \) has executed the associated event during \( t_i(\alpha, t) \), and 0 otherwise. Formally, let \( \gamma_i(t) \in \{ \mathcal{B}(t), \mathcal{C}(t), \mathcal{Y}(t), \mathcal{M}(t), \mathcal{A}(t), \mathcal{S}(t) \} \), then we have:

\[
 \gamma_i(t) = \begin{cases} 
 1 & \text{if } u_i \text{ has executed } \gamma_i \text{ till } t \\
 0 & \text{if } u_i \text{ has not executed } \gamma_i \text{ till } t
\end{cases}
\]

As shown in Table I, different events have different probability to be arisen. Hence, we consider the following weights indicating the impact of each event execution:

\[
 \omega_A = \frac{0.8}{3.6}, \omega_C = \frac{0.3}{3.6}, \omega_Y = \frac{0.4}{3.6}
\]

\[
 \omega_M = \frac{0.4}{3.6}, \omega_A = \frac{0.5}{3.6}, \omega_S = \frac{0.7}{3.6}, \omega_N = \frac{0.5}{3.6}
\]

Eventually, the effect of events in reliability reported by user \( u_i \) at time \( t \), denoted by \( \delta_i(\alpha, t) \), is computed as follows:

\[
 \delta_i(\alpha, t) = \omega_B \cdot \mathcal{B}(t) + \omega_C \cdot \mathcal{C}(t) + \omega_Y \cdot \mathcal{Y}(t) + \omega_M \cdot \mathcal{M}(t) + \omega_A \cdot \mathcal{A}(t) + \omega_S \cdot \mathcal{S}(t) + \omega_N \cdot \mathcal{N}(t)
\]

where \( \delta_i(\alpha, t) \in [0, 1] \), and the higher score shows that the report of user \( u_i \) about application \( \alpha \) is more reliable in terms of events evaluation.

Having the above functions and relations, is possible to define the reliability function, denoted by \( \mathcal{R}_i(\alpha, t) \), which returns a number in the interval [0, 1], such that the higher score means that the higher reliability is guaranteed on the report of user \( u_i \) about app \( \alpha \) at time \( t \).

Formally, we have:

\[
 \mathcal{R}_i(\alpha, t) = \begin{cases} 
 1 & \text{if } \delta_i(\alpha, t) + \max(0, \mathcal{R}_i(\alpha, t) - c) = \delta_i(\alpha, t) + \max(0, \mathcal{R}_i(\alpha, t) - c) = 1 \\
 0 & \text{otherwise}
\end{cases}
\]

where \( f_i(\alpha, t, \mathcal{P}) \), \( \tau_i(\alpha, t) \), and \( \delta_i(\alpha, t) \) are obtained applying relations 1, 2, and 3, respectively; and \( c \) is a constant real number in [0, 1], where the higher number means that the more weight is attributed to the impact of time without executing the events. In our experiments, we set \( c = 0.5 \). The amount of reliability equals to maximum output, i.e. 1, when the user report is based on the BRIDEMAID evaluation on her device. The reliability approaches to zero if the time duration of having application \( \alpha \) being active in the background of user \( u_i \)’s device goes to zero.

### E. Attacker Model

As the system is distributed, with testers being all peers, the lack of a root of trust exposes the framework to a set of attacks which have to be counteracted. We assume that attackers are able to completely modify the decision of BRIDEMAID, being thus able to submit their own decision for the app, with an arbitrary level of reliability. Being interested in pushing the system to accept their decisions, attackers will always choose a reliability score which is higher than the reliability threshold set by D-BRIDEMAID. A set of envisioned attackers is briefly presented in the following:
Reputation Tamperer: This malicious tester aims at tamp- ering the reputation of one or more genuine apps, in order to discourage users from downloading it. Reports submitted by this tester for genuine apps will deem the app as malicious with a high reliability level (i.e., higher than 0.8).

Malicious App Preacher: This tester accepts to participate to the evaluation program to push the system in considering as genuine an app which is malicious. This tester when submits reports for malicious apps, will always report the app as genuine, with an high reliability level.

Coin Flipper: The coin flipper gives random decisions on the trustworthiness of an app, with the objective of damaging the system still keeping a low profile to not be easily identified. This behavior can also be used to model users with a malfunctioning BRIDEMAID app.

Persistent Liar: This tester aims at maximizing the damage to the system, performing correct and reliable app analysis but producing always the opposite decision. This attacker is particularly dangerous if colluding with other attackers showing the same behavior.

F. Reputation Handling

The reputation model exploited in this work is based on the Jøsang model described in [9]. This model is based on a reputation score which is weighted by three components, namely belief, disbelief and uncertainty. The rationale behind choosing this specific model is the correspondence with the three possible actions that an UCS may perform when asked to provide a cached value. Every UCS has at the beginning a starting reputation score \( r_0 \). At each attribute reading in which a specific usage control system is involved, its reputation is updated according to the following formula: 
\[
    r(t) = b(t) - d(t) - u(t),
\]
where \( t \) is the time that the reputation is requested. Afterward, on the base of the provided value, after the decision process performed at step four of the system evolution, the three reputation component are updated according to the following algorithm.

The belief component is increased every time a user provides a value that is not wrong, i.e. the provided value is the one effectively considered good by the system. If the provided value is considered not reliable enough (less than 0.5), the uncertainty component is increased, whilst the disbelief component is increased if the provided value is considered reliable but wrong. No reputation changes happen for those users that choose to not provide any value. The values of \( \Delta_b \), \( \Delta_u \) and \( \Delta_d \) are configurable parameters. For the experiment performed in this work, the used values are: \( \Delta_b = 0.25 \), \( \Delta_u = 0.15 \), \( \Delta_d = 0.6 \), whilst the acceptance threshold for reputation \( \theta_r \) is set to 0.5. These values mildly increase at each reading the reputation of those users providing correct values, strongly penalize the users providing a malicious value, immediately reducing their reputation under the acceptance threshold. The uncertainty has a small impact on the reputation, which becomes consistent only after several non-enough-reliable readings.

G. Result Validity

With the definitions of reliability and reputation, it is now possible to formally define the decision process, used by our architecture to choose the boolean value \( V \in [0,1] \) for an application \( \alpha \) based on the reliability of what the users \( u_1, u_2, \ldots, u_N \) provide about this application and considering their reputation. The higher score of validity means that the associated application is more probable to be malicious. Moreover, let \( \theta_r \) and \( \theta_v \) be the thresholds of reliability and reputation, respectively.

To make the final decision, i.e. to give a malicious score to an application, the system collects the triples \( U_i(\alpha, t) = (f_i(\alpha, t), \mathcal{R}_i(\alpha, t), r_i(t)) \) for each user \( u_i \) who participated in the application evaluation, \( f_i(\alpha, t) \) is the value provided by user \( u_i \) about application \( \alpha \) at time \( t \), \( \mathcal{R}_i(\alpha, t) \) returns the reliability of \( f_i(\alpha, t) \), and \( r_i(t) \) is the reputation of user \( u_i \) at time \( t \).

The system discards the triples for which either \( \mathcal{R}_i(\alpha, t) \leq \theta_r \) or \( r_i(t) \leq \theta_v \). After discarding not reliable values, the set of \( n \) triples are considered as the following:

\[
U_i(\alpha, t) = (f_i(\alpha, t), \mathcal{R}_i(\alpha, t), r_i(t)) \quad \text{for} \quad 1 \leq i \leq n \tag{4}
\]

Then, from the collected information at time \( t \), the validity of application \( \alpha \), i.e. the malicious score of application \( \alpha \) based on the reports of \( n \) users, denoted by \( V(\alpha, t) \), is computed as the following:

\[
    V(\alpha, t) = \frac{1}{n} \sum_{i=1}^{n} f_i(\alpha, t) \times \left( \frac{\mathcal{R}_i(\alpha, t) + r_i(t)}{2} \right) \tag{5}
\]

where the higher output of \( V(\alpha, t) \) shows the higher malicious score of application \( \alpha \).

In this work, if \( V(\alpha, t) \geq 0.55 \), then the system reports the
application $\alpha$ as malicious. Moreover, if the number of reliable reports, i.e. reporting a reliability higher than $\theta_{\alpha}$, is lower than a configurable percentage of participating users, the evaluation is not considered valid and the app is scheduled for reevaluation.

### H. Revenue Computation

As incentive to testers to communicate correct and reliable reports, D-BRIDEMAID includes a revenue mechanism, which gives credits to testers for any usable report. A report, to be considered usable must match three conditions: (i) must be in line with the final decision of D-BRIDEMAID, (ii) should be reliable, i.e. the report reliability should be higher than $\theta_{\alpha}$, (iii) the tester reputation should be higher than $\theta_r$. Hence, for the testers matching these conditions, the reputation increment at the end of the evaluation epoch will be computed as follows:

$$\rho_{\text{inc}} = \begin{cases} 
\rho (r_i - \theta_r) & \text{if } r_i \geq \theta_r \\
0 & \text{otherwise}
\end{cases}$$

where $\rho$ is a configurable parameter and $r_i$ is the reputation of the tester for a specific report. It is worth noting that the higher is the reputation, the higher will be the received revenue.

### IV. EXPERIMENTAL RESULT

To validate our methodology simulative experiments have been run to simulate the interactions among a set of 50 testers participating at the evaluation campaign of 100 apps, 20 of which genuines and 80 malicious. Simulative experiments have been run on a custom simulator, allowing to vary the percentage of malicious users for each run. Each run for each configuration has been run 1000 times with different seeds, hence reported results are in the form of average and standard deviation. It is worth noting that no Android apps have been effectively run in the simulator, instead the behavior of each app has been set as simulation parameters, specifying the possibility for the malicious code to activate for any of the actions reported in Table I.

#### A. Simulative Experiments

In the simulation for each evaluation epoch, a random number of testers ranging between 30 and 50 choose to participate to evaluate that specific app. Table II summarizes all the parameter values set for the simulator. As discussed experiments have been run varying both the attacker percentage and aggressiveness. This last parameter, in particular, characterizes the behavior of attackers, stating their percentage of malicious reports among all the ones that release during the simulation. The rationale behind this parameter is the possibility of attackers wishing to conceal their actions, presenting some correct reports in order to increase their reputation before performing the attack. All simulations have been run with the four distinct attacker models described in Section III-E, i.e. all attackers of any simulation behave according to the same model. From all experiments the extracted results are the reputation for good and malicious testers, their revenue and the percentage of the apps correctly classified by D-BRIDEMAID.

### Table II: Simulation Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tester Number</td>
<td>50</td>
</tr>
<tr>
<td>Apps</td>
<td>100</td>
</tr>
<tr>
<td>Genuine Apps</td>
<td>20</td>
</tr>
<tr>
<td>Malicious Apps</td>
<td>80</td>
</tr>
<tr>
<td>$\theta_r$, $\theta_{\alpha}$</td>
<td>0.5, 0.7</td>
</tr>
<tr>
<td>$\Delta_{\alpha}$</td>
<td>0.25, 0.6, 0.15</td>
</tr>
<tr>
<td>$\rho$</td>
<td>2</td>
</tr>
<tr>
<td>Number of cycles</td>
<td>1000</td>
</tr>
<tr>
<td>Attacker percentage</td>
<td>10% $\leq x \leq$ 50%</td>
</tr>
<tr>
<td>Aggressiveness</td>
<td>10% $\leq x \leq$ 100%</td>
</tr>
</tbody>
</table>

#### B. Results

Table III schematically reports the classification results of D-BRIDEMAID at the variation of the aforementioned parameters. In particular the table shows the percentage of apps correctly classified, according to the received reports. Results are reported by varying the percentages by 10% at each experiment. As shown for the majority of configurations, the percentage of correctly classified apps is higher than 90%. There is a degradation of the accuracy of the framework only

### Table III: Percentage of apps correctly classified.

<table>
<thead>
<tr>
<th>Aggressivity</th>
<th>PM 10%</th>
<th>PM 20%</th>
<th>PM 30%</th>
<th>PM 40%</th>
<th>PM 50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>20%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>30%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>40%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>50%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>60%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>70%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>80%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>90%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Aggressivity</th>
<th>PM 10%</th>
<th>PM 20%</th>
<th>PM 30%</th>
<th>PM 40%</th>
<th>PM 50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>20%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>30%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>40%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>50%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>60%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>70%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>80%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>90%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Aggressivity</th>
<th>PM 10%</th>
<th>PM 20%</th>
<th>PM 30%</th>
<th>PM 40%</th>
<th>PM 50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>20%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>30%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>40%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>50%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>60%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>70%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>80%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>90%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
when the malicious tester colludes and they are a consistent percentage of all testers. In particular the attackers are able to defeat the good users exploiting the Persistent Liar attacker model, with a strong aggressiveness (higher than 60%) and when they sum to more than 30% of all users in the system. This is due to the fact that normal users may, provide wrong reports, or reports with a low reliability that are not taken in consideration by the system, whilst the malicious users are colluding to provide a fake report. We recall, in fact, that malicious users always show high reliability in their reports. It is possible to notice a low accuracy also for the case of Malicious App Preacher and Reputation Tamperer. Again this is due to the high reliability given to these reports which become more influential in the case of malicious app, that, by configuration, are 80% of the share of all apps. It must be noticed, however, that the reported performance support the validity of the proposed model, which in the worst setting, is able to be resilient up to 20% of malicious and colluding attackers. Results for reputation and revenue with different attacker models are reported in Figure 3. In Figure 3a it is represented the average reputation pattern for good and malicious testers, on the evaluation of 100 apps, where all attackers follow the persistent liar model. For the sake of clarity have been reported only the results related to the experiments with 10% and 20% as percentage of malicious users. The reputation results confirm what shown in Table III, in fact, it is possible to see that reputation of malicious users immediately drops under the threshold, thus not giving the possibility to attackers to send reports considered in the evaluation process. Concerning the revenue (Figure 3c) it is possible to see that attackers do not receive any revenue for the first two experiments, whilst they become able to control the system, thanks to collusion if they are more than 30% of selected testers. Figure 3b and Figure 3d shows reputation and revenue when the attackers use the coin flipper attack model. As shown, the reputation of coin flipper always remains for most of the time under the threshold of 0.5, for all percentages up to 50%. The earned revenue is thus negligible and the system is slightly affected even when the percentage of attackers reaches 50%.

V. RELATED WORK

Oberheide et.al. proposed CloudAV [10], a system where end hosts send suspicious files to a central cloud-based antivirus service for scanning by a number of different AVs. A threshold approach is used to aggregate feedback from multiple antimalware. An implementation of CloudAV is described in [11]. RAVE [12] is another centralized collaborative malware scanning system where emails are sent to several agents for malware scanning. A voting based mechanism is employed to make final decisions. In RevMatch [13] collaborative malware decisions are made based on labeled malware detection history from participating antimalware. Their model is evaluated using real-world malware data sets and demonstrate that collaborative malware detection techniques can improve the malware detection accuracy compared to using a single albeit the best antivirus. Social-AV [14] leverages two key
ideas: social collaboration and the concept of a hot set. The hot set concept states that not all malware signatures are equally important. At any given time, some signatures (i.e., the hot set) are more likely to be matched than the others. Social-AV only keeps the hot set of signatures in the main memory, and distributing the whole signature database among devices belonging to the social group of the device owner. The main limitation of these approaches is represented by the usage of antimalware, as matter of fact signature provided by free and commercial antimalware are easily evaded by zero-day attacks or using trivial code obfuscation techniques, as demonstrated in [2]. Buenenmeyer et al. [15] present an approach for monitoring current changes on a smartphone in order to detect anomalies. The changes can be caused by malwares and external attackers, e.g., flooding or network probing. The monitored data is sent to a remote server that creates profiles of each monitored device in order to detect anomalies. The main difference with our method is represented by the fact that this approach focused on network traffic (by WiFi, bluetooth) to detect anomalies and it is applied to devices with Windows Mobile on board. NetBuckler [16] is a collaborative client application which employs collaborative intelligence to defend against Internet worms. NetBuckler creates a peer-to-peer network where peers can meet in custom peer groups and communicate traffic related security information, and/or locally enforce security measures depending on the information received. Our work is different because is related to Android environment. A collaborate mechanism proposal to anticipate Advanced Persistent Threat is described in [17]. The paper presents a design to detect zero day attack using windows function hooking that might help the information security community to detect such malicious attacks well in time so the appropriate defensive actions could be taken. This method is able to intercept the invocation of malicious DLLs relating to Microsoft Windows environment, while our method is related to Android OS.

VI. CONCLUSION AND FUTURE WORK
In this paper we have presented D-BRIDEMAID, a collaborative and distributed framework for dynamic analysis of malicious Android apps. The presented model, based on a peer-to-peer architecture, and algorithms to compute report reliability and tester reputation with a punish/reward model, proves to be resistant to different attacker models, including colluding attackers. In fact, D-BRIDEMAID is able to correctly recognize genuine reports, till the percentage of attackers is under a a threshold. As future work, we plan to extend the model to make the system more resilient to colluding attackers, with strategies based on received reliability, to identify malicious testers. Additional experiments considering the evaluation of real apps, with a direct monitoring of the on-host IDS performances will also be implemented.

ACKNOWLEDGEMENTS
This work has been partially supported by H2020 EU-funded project European Network for Cyber Security (NeCS), GA #675320 and EIT-Digital Project MCloudDaaS.

REFERENCES