A recommender-based system for assisting non technical users in managing Android permissions

Arnaud Oglaza, Romain Laborde, François Barrère, Abdelmalek Benzekri

A recommender-based system for assisting non technical users in managing Android permissions

Arnaud Oglaza
Toulouse France
Email: contact@kapuer.org

Romain Laborde
University of Toulouse
IRIT UMR 5505
Toulouse France
Email: laborde@irit.fr

Abdelmalek Benzekri
University of Toulouse
IRIT UMR 5505
Toulouse France
Email: benzekri@irit.fr

François Barrère
University of Toulouse
IRIT UMR 5505
Toulouse France
Email: barrere@irit.fr

***Abstract*—Today, permissions management solutions on mobile devices employ Identity Based Access Control (IBAC) models. If this approach was suitable when people had only a few games (like Snake or Tetris) installed on their mobile phones, the current situation is different. A survey from Google in 2013 showed that, on average, US users have installed 33 applications on their Android smartphones. As a result, these users must manage hundreds of permissions to protect their privacy. Scalability of IBAC is a well-known issue and many more advanced access control models have introduced abstractions to cope with this problem. However, such models are more complex to handle by non-technical users. Thus, we present a permission management system for Android devices that 1) learns users' privacy preferences, 2) proposes them abstract authorization rules, and 3) provides advanced features to manage these high-level rules. We prove this approach is more efficient than current permission management system by comparing it to Privacy Guard Manager.**

I. Introduction

Android software development kit allows developers to access hardware features of Android devices like GPS location or the camera but also users' data such as contact lists or calendars. To actually have access to each feature, developers have to request the associated permissions in a manifest file. On versions 4.4 and 5, which are today the most deployed, users are informed of the permissions an application requires during the installation. At that time, they only have one choice: accept to provide the application with unlimited access to all the requested permissions or decline and cancel the installation.

Additional permission management systems (such as Privacy Guard Manager, Permission Master, XPrivacy or DonkeyGuard) can be installed to enhance the basic native Android system by allowing users to modify permissions after the installation of applications. All these permission management applications follow an Identity Based Access Control model, i.e., the user has to control every permission for every installed application. Although IBAC allows fine-grained access control, it is not suitable for managing hundreds of permissions. Scalability issue has been studied by the access control models research community that proposed the use of abstractions leading to high-level authorization rules making global access control policies more understandable. Nonetheless, applying these models requires to understand the associated abstractions and thus suffer accessibility for non-technical users.

More than just a need, privacy is a right [1]. We must give users a way to control the disclosure of their private data. In this article, we present a recommender-based system, called Kapuer (KAPUER is an Assistant for Protection of Users pErsonnal infoRmation), that assists people in managing permissions on their Android device. Kapuer includes a machine learning algorithm, which extends the study performed in [2], to capture users' preferences in terms of privacy. These preferences when validated by users, are transformed into XACML V3 policies. We developed a plug-in based on the Xposed[1] framework that enforces the XACML policies. Kapuer also includes additional permission management features to enhance its efficiency to understand and visualize abstract authorization rules.

Kapuer is freely available to download at the following address: http://www.kapuer.org.

The rest of the article is structured as follow: first, we review and discuss access control management approaches applied to Android. In Section 3, we present the generic architecture of Kapuer as well as its problem-solving model for the machine learning algorithm. Section 4 details our Android implementation. We analyze available security information related to Android permissions and propose hierarchies of criteria for Android. We evaluate Kapuer in Section 5 on a real life scenario. Finally, we conclude in Section 6.

II. Related works

In this section, we summarize different works related to Android permission management.

A. High level policies in Android

Access control models can be seen as design patterns to help the specification of policies. They all consider three main entities: the subject, the action, and the resource. In addition, some access control models propose abstractions. For Barker [3], these abstractions (he calls them categories) represent "any of several fundamental and distinct classes or groups to which entities may be assigned". One of the advantages of

[1] http://repo.xposed.info/

using abstractions is simplifying policies. For instance, Role Based Access Control (RBAC) [4] uses the concept of role to group subjects according to their function in an organization. Other access control models propose abstractions for other elements. Organization Based Access Control (OrBAC) [5] uses abstractions on all three main elements: roles abstract subjects, activities are for actions and views for resources. Some access control models are designed for privacy like PBAC [6] that introduces the intent of the subject or P-RBAC [7] that extends RBAC with concepts like purpose, condition, and obligation. An exception is Attribute Based Access Control (ABAC). ABAC does not introduce abstraction but is a specification pattern to express authorization policies using any abstraction.

Each model offers concepts and abstractions to guide security experts in the writing of policies. Even if analyzing and implementing abstractions is time-consuming, resulting authorization policies are more powerful and easier to manage. But it requires security expertise for writing authorization policies and most of mobile device users lack this skill.

Some systems already use access control models to enforce Android permissions. For instance, CRêPE [8] uses context-related policies to control how applications can use their permissions. Detection of contexts triggers the activation of the policy to use. MOSES [9] is also a system enforcing policy based security on Android. MOSES relies on system compartmenting to create isolated areas. Each area can be used to separate, for example, data and applications used for work and those for personal use. Although both CRêPE and MOSES allow Android to enforce high-level policies, they did not address the usability issue. Only skilled people can write those policies or define contexts and security profiles.

B. Writing policies with a graphical editor

Until version 5.x, official Android releases did not provide any efficient tools to manage permissions. Many custom releases or applications give users a way to better control permissions (such as Privacy Guard Manager, Permission Master, XPrivacy or DonkeyGuard). Our study will concentrate only on Privacy Guard Manager (PGM) from CyanogenMod[2] since all other applications have the same drawback. PGM is available in the settings menu of CyanogenMod and presents all installed applications. For each application, there is the list of its permissions. Users have the choice to select ON or OFF for each application to allow or deny the permission. An option also exists to ask users to decide at the first time the permission is requested. Thus, the interface is very easy to master and requires no technical skills: only a simple action on the device for each permission.

Although the process is easy to learn, checking all permissions for every applications is painful. The survey from Google[3] in 2013 shows that a US smartphone's user has an average of 33 applications installed on his device. We performed an analysis of the 50 most downloaded free applications on the Google Play Store and found that an application asks for an average of 11.4 permissions. This gives us a total of 376 permissions. Few users will browse the whole list of permissions to protect their privacy. Although PGM is interesting when dealing with few applications, current smartphone environment is much more complex and using no abstraction has difficulties facing scalability.

Google has improved its native permission management approach in the latest version of Android (version 6.0). First, permissions are requested at runtime (the first time applications require them) and users can change allowed/refused permissions at any time. Now, Android application developers shall manage the fact that their application may not have access to all the permissions listed in the *manifest file*. Google has also addressed the large number of permissions issue by introducing protection levels and groups of permissions. Each permission is associated to one of four protection levels. Permissions with level *normal* are considered as low-risk permissions and are automatically granted without any user approval. Permissions with level *signature* are related to communication between applications developed by the same organization. The requesting application needs to be signed by the same certificate as the application providing the service and declaring the permission. In that case, the system automatically grants the permission without any action required from the user. Protection level *signatureOrSystem* works like *signature* but concerns also the applications that are in the Android system image. Finally, the last level, *dangerous*, contains permissions with high risk for the user. There are 24 *dangerous* permissions. To reduce this number, every *dangerous* permission is attached to one permission group. Nine different groups exist and each one of them represents a resource or a set of resources like *CALENDAR*, *CONTACTS* or *PHONE*. For example, the group *CALENDAR* consists in two permissions *read calendar* and *write calendar*. To reduce the number of interactions with the user, the new permission management works as follows. When an application requests a *dangerous* permission, Android does not ask the user to accept or deny that particular permission. It asks the user to accept or deny the whole permission group.

Thus, even if there are more than 130 permissions in Android, a user will be asked to grant or refuse permissions to a specific application only nine times at most (one per group). Although this approach seems more user-friendly, it has significant drawbacks in terms of security. All permissions to access any network services (3G/4G, NFC, Bluetooth) are associated to protection level *normal*. As consequences, users cannot control network access and any application can communicate anywhere. In addition, this loss of control is increased by the use of groups of permission. Indeed, users' control regresses even with *dangerous* permissions. For instance, group *PHONE* includes seven permissions (*use SIP*, *call phone*, *read phone state*, *process outgoing calls*, *read call log*, *write call log* and *add voicemail*). Thus, when a Voice over IP application requests permission *use SIP* which seems relevant, the user can

[2]http://www.cyanogenmod.org/

[3]http://think.withgoogle.com/mobileplanet/fr/

only grant the group of seven permissions. These permissions are not all relevant to a VoIP application. Thus, to limit the number of interactions with the users and cope with scalability, Android 6.0 has decreased the privacy protection capability of the system. These IBAC coarse-grained permissions only give the user an illusion of control.

C. Writing policies with a text editor

Textual editors use specific languages to write authorization policies. Textual editors are less accessible than graphical ones because the language must be learned and understood before writing anything. However, these languages provide much more flexibility and the possibility to create very powerful rules.

XACML V3 [10] (eXtensible Access Control Markup Language) is a language standardized by OASIS, for writing authorization policies. XACML uses attributes to build policies thereby works great with ABAC. Every security element can be represented as an attribute. Then it is possible to create any kind of abstractions such as roles in RBAC, activities in OrBAC, etc. Genericity and flexibility are the main advantages of XACML but also its main flaw. Technical skills are required: understanding access control models to select suitable abstractions and write policies according to them. In addition, XACML is an XML language which is not known to be user-friendly for non-technical people. Thus, the whole process demands lots of technical skills and cannot be performed by owners of smartphones.

Arena et al. [11] have proposed an XACML-based extension of the Android's security framework called SecureDroid. This tool allows users to define situations and specify which permissions are accepted or denied in these situations. Users can also be prompted when a permission is requested. This approach do not use abstraction so scaling up is still a problem. Stepien and al. [12] have worked on a graphical editor to help non-technical users write XACML rules. With this editor, it is possible to choose an attribute, an operator and a value to compare to. It makes writing rules possible without using an XML format. Nonetheless, understanding how abstractions work is still required.

III. A DECISION SUPPORT SYSTEM FOR WRITING HIGH LEVEL POLICIES

Allowing non-technical users to write policies by themselves is not a simple task. A graphical editor like PGM is easy to use but lacks efficiency. With hundreds of permissions to handle, using abstractions seems mandatory. These abstractions can be specified with textual editors but it requires a lot of technical skills and then it is not accessible to the public. Non-technical users should be helped by a security expert to write authorization policies to protect their privacy but, of course, it is not possible to have an expert behind every smartphone user. Since none of these approaches is satisfying, we present our work which aims at: i) *requiring no skill before being used like PGM*, ii) *allowing non-technical users to write policies with abstractions*.

We have chosen to create a Decision Support System (DSS) to help users to write their complex policies [13]. DSS are a set of methods and techniques used to help someone facing a problem to make a decision [14]. We use a DSS to interact with users and understand how they wants to protect their data. We present in this Section our system, named Kapuer, applied to Android permission management. It informs users when applications request permissions, it learns how users react to requests and it uses these preferences to propose abstract authorization rules. Kapuer consists in an architecture to interact with users and control applications, and a problem-solving model to learn users' preferences.

A. Introduction to Decision Support System

The main goal of a Decision Support System is *not to make the decision on behalf of the user but instead to give him precious information to understand the situation, to give parts of solutions or possible alternatives to allow him to make the final decision* [14]. Among the different approaches of DSS, we have focused on recommender systems. These systems work with a profile of the user. It filter and analyze information, extract the most useful to build knowledge about users, their preferences. By learning these preferences, the system is able to propose solutions to the user by analyzing new information each time new preferences are acquired. Thereby, the system is always learning and adapting itself to the user. Three types of recommender systems exist [15]:

1) Content-based recommendations rely only on objects characteristics to make propositions. All available information on the object can be used to describe it. For example, a book can be described by a title, an author, a release date, etc. To make recommendations, the system compares objects to find those that seems to be the closest to the user's preferences. Content-based recommendations are very interesting with detailed objects. Because they are seen as a set of characteristics, a new object can be immediately proposed to users if its characteristics fit their preferences. If users have always the same behavior, the system will always propose relevant objects. The drawback is the starting: when the system has no information about user's preferences, a learning period is required before propositions can be relevant. Similarly, if users suddenly changes their behavior, there will be a certain latency before the system learns those changes.

2) Recommendations by collaborative filtering work with the preferences of all people using the system. The idea is if one user has similar preferences with other users, then he should like objects chosen by such users. Thereby, they can be relevant recommendations for him. Unlike content-based, the system does not need much information to start. It will quickly find other people with a close profile. Collaborative filtering also works fine with objects that are hard to describe like emotions. This approach also has some drawbacks. When few people are using the system, finding a similar profile might fail. In this case, recommendations will not be relevant. In the same way, if a new object is added to the system, as long as it is not chosen by some users, it will not be recommended.

3) Hybrid systems uses both content-based and collaborative filtering. It allows getting rid of some flaws of each approach. Content-based recommendations for new objects in the system, collaborative filtering for users with few information to work with. A well known hybrid recommender system is used by Amazon to create lists of similar items when a customer visits the page of an object or adds one in his basket.

Despite the hybrid recommender system advantages, we chose a pure content-based approach in Kapuer for two reasons. Firstly, using collaborative filtering requires storing every user's privacy preferences somewhere on a server. Protection of users' preferences is complex [16][17]. With a content-based recommender system, user's preferences are stored locally and are not shared at all. Secondly, privacy recommendation can also be provided by a set of experts like in [18]. We think that privacy is by nature personal and these solutions do not allow experts to customize their recommendations to specific users. E.g., the four authors of this article do not agree on what access should be granted to the Facebook app.

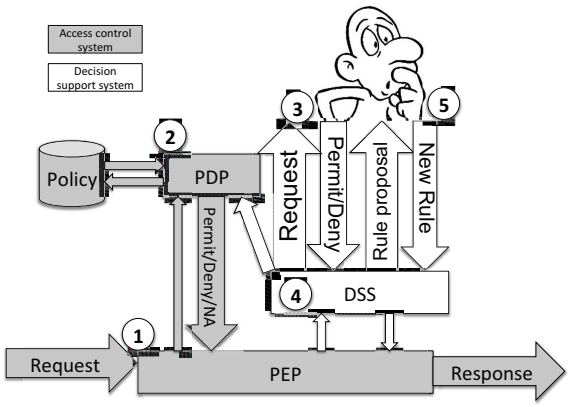

Fig. 1. The architecture of Kapuer

B. Architecture of Kapuer

Figure 1 shows the architecture of Kapuer with a clear distinction between the XACML access control part and the decision support part. The process begins when an application requests the access to one of the user's private information. This request is intercepted by the Policy Enforcement Point (PEP) (step 1) that creates a request in the XACML V3 format and transmits it to the Policy Decision Point (PDP) (step 2). The PDP is the decision unit; it compares the request with the access control policy. If one of the rules matches, the PDP sends back the associated decision (Permit or Deny) to the PEP which applies it on the Android system. If there is no matching, the PDP returns a "Not Applicable" decision to the PEP. No matching rule means that our system needs to learn more about the user's preferences about this request. As a consequence, the PEP transfers the request to the DSS. To get more information about the user's preferences, the DSS interacts with him. Kapuer has two sorts of interactions with the user:

1) When a request is sent by the PEP to the DSS (step 3), Kapuer informs the user that an application wants to access one of his private data. If possible, details about the request are also given such as similar applications asking for the same permission or indications about the permission asked. The user just needs to accept or decline that request. Making a decision in the appropriate context (i.e. when the user runs the application) is easier than doing it out of the context of use; it limits the cognitive load. Once the user has made his decision, the DSS creates the corresponding XACML V3 rule for that specific couple (application, permission) and puts it in the policy database (step 4). Then, all the preferences regarding the attributes used in the request are updated (more information on the preferences update can be found in [2]). When the update process is completed, Kapuer calculates a score reflecting the user's preferences knowledge level.

2) When the score of a request reaches a predetermined threshold (found by experimentations), the system has acquired enough information to propose a new abstracted rule that covers a broader range of access requests. In this interaction, Kapuer presents this rule to the user and explains all the abstract elements (step 5). For example, if a rule is proposed for all game applications, Kapuer lists all games. If a rule is proposed for all resources linked to networks, it details all the associated permissions. This way, user are informed and even without any technical skills, they are able to understand what the rule means. Then they can accept or reject the proposed rule. If accepted, the DSS transforms the rule in XACML V3 and adds it to the policy database (step 4 again).

C. Our problem solving model

The main objective of Kapuer is learning users' preferences to recommend them high-level rules. The DSS extracts data from requests and analyzes them with the user's decision. In order to explain how Kapuer learns the users' preferences, we must first present our problem-solving model. This model is independent of any access control model and is based on four elements :

- **Criteria** - A criterion is the basic and main element of our problem-solving model. It represents an attribute of an access request like the name of an application, a resource or an action. We have defined criteria to be very similar to attributes in ABAC. An attribute-based request can be easily converted into a list of criteria. The set of criteria is noted CR. Criteria are composed of an identifier and two values. The first value, $g^t : CR \rightarrow [0, \infty[$, increments each time the user accepts a request including this criterion. The second value, $f^t : CR \rightarrow [0, \infty[$, increments each time the user refuses a request including this criterion. The preferences score regarding this criterion can be found by subtracting f^t and g^t. This dual unipolar scale gives much more information than a unique bipolar scale. We can easily differentiate if a criterion has a low preferences value because we have few information on

user's preferences or because users do not always behave in the same way (sometimes they accept requests with this criterion and sometimes they deny them).

- **Classes of criteria** - We introduce the notion of class of criteria to express security objects introduced in the access control models like visibility, temporal and spatial aspects, retention or purpose. Each criterion is part of a class with relation Association Criterion Class: $ACC \subseteq CR \times C$ where the set of class of criteria is noted C. It is possible to create any class depending on the type of data the system uses. The set of criteria of a class is defined by the function *class* :

$$class : C \rightarrow \mathcal{P}^{CR}$$
$$x \mapsto \{y \in CR | (y, C) \in ACC\} \quad (1)$$

- **Meta-criteria** - Access control models propose abstractions of security objects. We define the notion of meta-criterion to represent these abstractions. For instance, "Games" is a meta-criterion for applications describing this kind of application. A meta-criterion is a criterion, with the same structure but with a higher level of abstraction. The set of meta-criteria is noted MCR where $MCR \subset CR$. A meta-criterion is also part of a class. Each criterion is linked to one meta-criterion of the same class. A meta-criterion can be linked to another meta-criterion of a higher level. Then, we can create a hierarchy of criteria and meta-criteria for each class. Values of meta-criteria are updated each time a criterion or a meta-criterion linked to it is updated. Then if a criterion is updated, all meta-criteria in the same branch of the hierarchy are also updated.
- **Groups of criteria** - We have defined groups of criteria to represent relations between criteria or meta-criteria from different classes. A group of criteria is formed by at least two criteria and at most by the number of classes. A group has his own preferences values, they are not calculated from the values of the criteria or meta-criteria present in the group. Values of groups are updated each time all the criteria or meta-criteria of the group are present in a request. The larger a group is, the more detailed information about users preferences it gives. Thus, larger group are more important. The set of groups of criteria G is defined by:

$$G \subseteq \mathcal{P}^{CR}$$

A group of criteria is composed by at least two criteria.

$$\forall g \in G, |g| \geq 2$$

Two criteria of a same group cannot belong to the same class.

$$\forall g \in G, \forall (c_1, c_2) \in g \times g, c_1 \neq c_2 \Rightarrow class(c_1) \neq class(c_2)$$

This solving problem model is implemented by our aggregation operator, called Kagop, to calculate scores of requests and propose abstract rules (more information on Kagop are available in [2]).

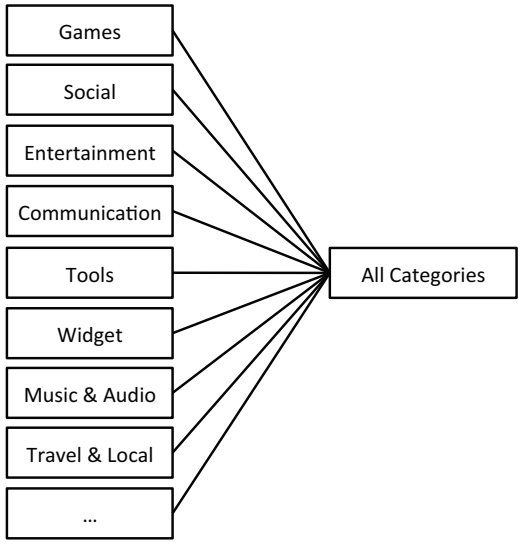

Fig. 2. Hierarchy of criteria for class Applications

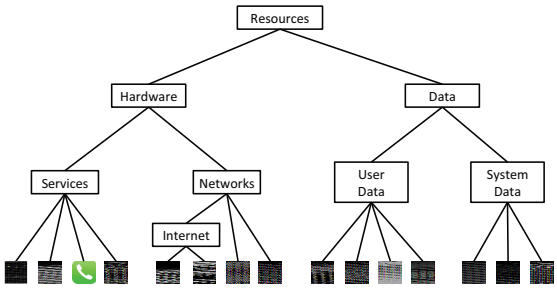

Fig. 3. Hierarchy of criteria for class Resources

IV. The Android implementation

Due to Android 4.4 security protections, it is not possible to intercept permission requests in an application, they must be intercepted directly from the OS. It has forced us to use the framework XPosed which allows modifying the source code of Android without the need to make a custom release. With the help of XPosed and the Balana[4] implementation of XACML V3, we built the module Kapuer for Android 4.4. We describe in this Section the instantiation of our problem-solving model to Android, the interactions during the learning phase and finally abstract authorization rules management functionalities.

When Kapuer intercepts a permission request, information about the application and the permission is collected. Thus, we have defined three classes to implement our problem solving model: *Applications*, *Actions* and *Resources*. Criteria of class *Applications* are represented by the name of the applications installed on the device. Kapuer gets the first level of meta-criteria from the application categories provided by the Google Play Store (e.g., games, entertainment, work, etc.). We have created a meta-criterion called *no category*

[4]https://github.com/wso2/balana

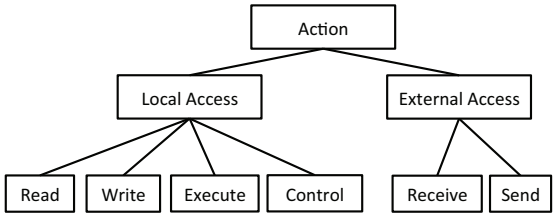

Fig. 4. Hierarchy of criteria for class Actions

for applications not included in the Google Play Store. Finally, we added meta-criterion *all applications* to group all the meta-criteria of this class (Figure 2). Criteria of classes *Actions* and *Resources* are extracted from the permissions (e.g., "android.permission.READ_CALENDAR" contains criteria READ and CALENDAR). We chose for class *Resources* to use meta-criteria *media*, *network*, *service*, *user data* and *system data*. We added meta-criteria *hardware* (superior to *media*, *network*, *service*) and *data* (superior to *user data* and *system data*). Finally, the root of this hierarchy is meta-criterion *all resources* (Figure 3). With the same reasoning, we have extracted criteria from the permission list for class *Actions*. We have defined three meta-criteria : *local access* (regrouping criteria *execute*, *control*, *read* and *write*), *external access* (regrouping criteria *send* and *receive*) and the root of the hierarchy called *all actions* (Figure 4).

Figures 5 and 6 shows the interactions with the user during the learning phase. Interaction in Figure 5 is displayed when Kapuer has no rule to handle the request and asks the user to make a decision (corresponds to the step 3 in Figure 1). Interaction in Figure 6 is displayed when Kapuer has enough information on the user's preferences and proposes a high-level rule (step 5 in Figure 1). If offers information about all abstractions used in the rule. In this example, there are only meta-criteria so we detail all criteria contained in each meta-criteria to help the user understand exactly what this new rule will do if the user accepts it.

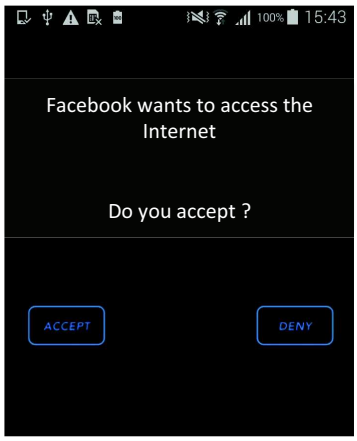

Fig. 5. Interactions of Kapuer during the learning phase

Kapuer offers also an interface for viewing the privacy policy and features to modify it. Figures 7, 8 and 9 shows three screenshots of the application. Users can access the list of all rules and have a description for each one (Figure 7). They can also edit each rule and modify it on four parts: the application, the resource, the action or the decision (Figure 8). Except for the decision, they can modify the level of abstraction if needed. We added a summary of all permissions granted to an application (Figure 9). This view provides detailed information about permissions handled by Kapuer.

V. Evaluation of Kapuer for managing Android permissions

We evaluate in this Section Kapuer's efficiency. Firstly, we compare it to PGM in a real scenario. Secondly, we evaluate the speed of the learning process.

A. Kapuer VS Privacy Guard Manager

We wanted to test the approach with a real life situation. First, we installed the 50 most downloaded free applications in the Google Play Store. It resulted in 28 games, 4 social apps, 4 communication apps, 4 widgets apps, 3 tools, 3 entertainment apps, 2 apps about music and 2 apps about travel. Then, we defined the following arbitrary high-level authorization policy we want to enforce on our Android device:

- *rule 1*: Games can access the Internet.
- *rule 2*: Social applications can access network and system data.
- *rule 3*: Communication applications can access network and services.
- *rule 4*: Widget applications can access the Internet.
- *rule 5*: Music&Audio applications can access network and audio.
- *rule 6*: Tools applications can access everything.
- *rule 7*: Travel&Local applications can access network and GPS.
- *rule 8*: Entertainment applications can access the Internet.

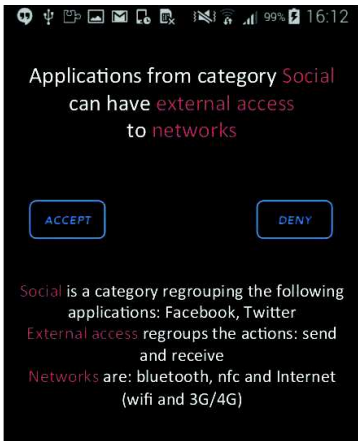

Fig. 6. Proposition of a new high level rule

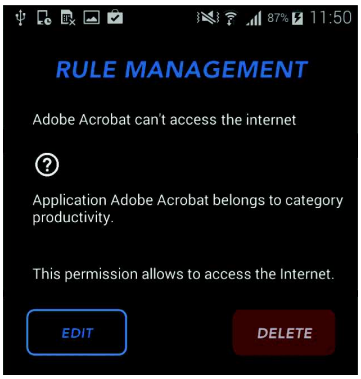

Fig. 7. Rule information screen

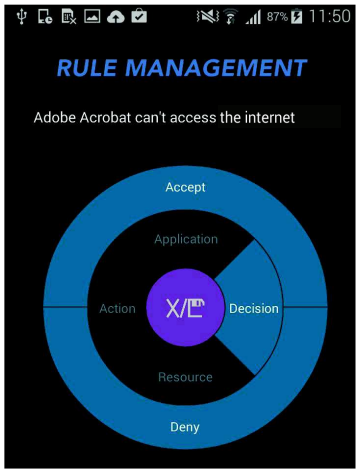

Fig. 8. Rule modification screen

We want to evaluate the cost of writing this high-level policy using PGM and Kapuer. This cost is calculated by the number of actions the user has to perform. For PGM, an action consists in all pressures (screen navigation and on/off flipswitches selection). For Kapuer, any interaction as explained in Figures 5 and 6 is an action. Since Kapuer learning process is not predetermined and depends on the received requests, a large number of tests must be executed to get its average behavior. Thus, we used our simulator [19] that can automate this task. This simulator is able to generate random requests based on a set of possible accesses. It also automates the user behavior by accepting or denying requests based on a predefined high-level policy. We ran 10 simulations with the same high-level policy but with different requests each time since they are generated randomly. After each simulation, we checked the rules recommended by Kapuer.

Figure 10 illustrates the number of actions needed to write each rule of our privacy policy. It also shows that Kapuer needs fewer actions than Privacy Guard Manager for each rule. There is even a huge difference on the first rule. It concerns all the applications with the category “Games” and they represent 28

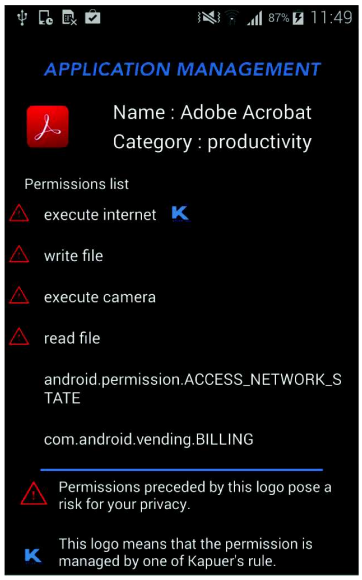

Fig. 9. Application information screen

applications out of 50 so more than the half. For this rule, Kapuer needs 61 actions and PGM 476, so nearly eight times more. For rules number 2 and 3, the difference between Kapuer and PGM is also important when it is closer for all the others. This is due to the few number of applications involved in the last rules. If we compare the whole policy, Kapuer required 190 actions only when PGM has needed 848 actions. The abstractions in the high-level rules proposed by Kapuer are really providing a faster process for the user to fulfill his privacy policy. We have also looked at these high-level rules in details to see if some of them do not fit the user’s preferences. None of the created rules proposed the opposite of what the high-level policy stated. Nevertheless, some of them did not have the right level of abstraction. It happens that a rule is proposed to the user with a higher abstraction than needed so it does not totally fit the user’s behavior.

B. Evaluation of the speed at which Kapuer learns privacy preferences

The number of interactions needed to recreate one rule or all the policy provides information about the effort needed by users. It is also interesting to see how fast Kapuer is learning and how the level of completeness progresses. We compared, for each simulation, how many requests were needed to reach, 20%, 50%, 80% and 100% of completeness. The results are shown in Figure 11. At the beginning, Kapuer does not know anything about the user’s preferences. It needs requests to learn his behavior and to start proposing high-level rules. The first rule is proposed on average after 50 requests. Then, the average number of requests to reach the first threshold, 20% of completeness, is 75. For the second threshold, 50%, the average number of requests is 104. For the 80% threshold, the average number of requests is 123 and finally 100% of

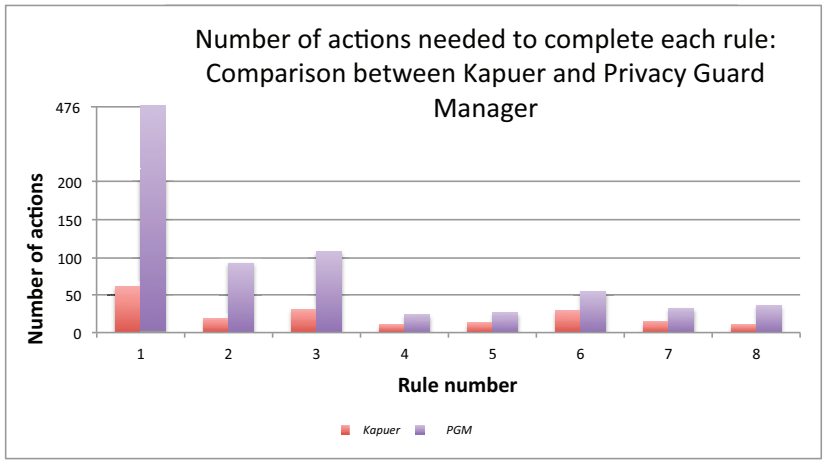

Fig. 10. Comparison of Privacy Guard Manager and Kapuer

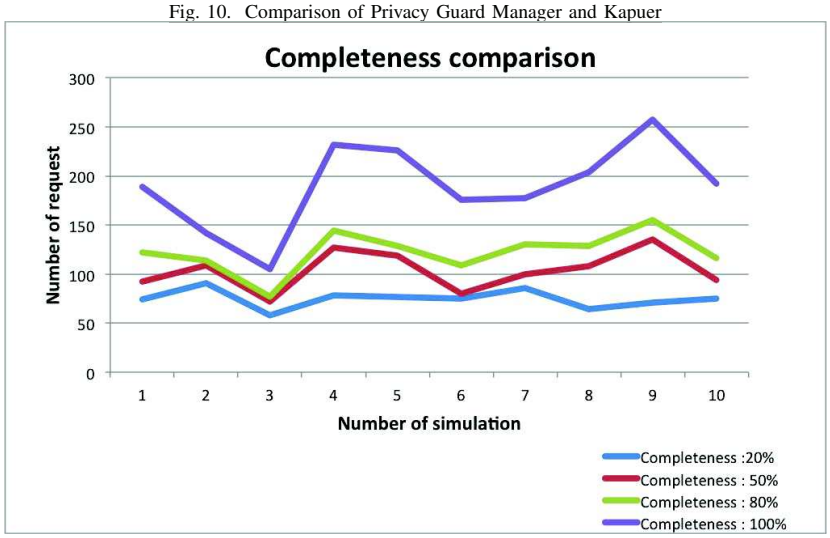

Fig. 11. Completeness for each rule

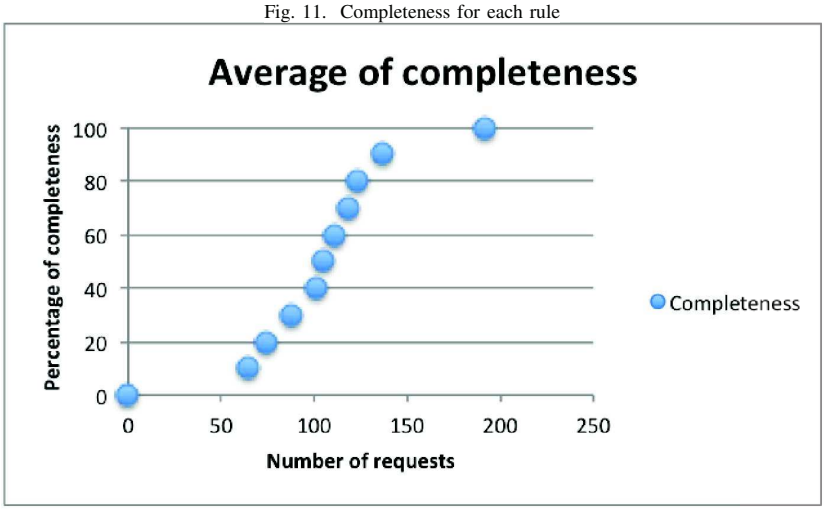

Fig. 12. Average of completeness

completeness needs an average of 190 requests. From these simulations, we calculated the global average completeness. Figure 12 shows the average of completeness at 10%, 20%, 30% etc. It confirms that after learning from the firsts requests, Kapuer proposes regularly rules until the policy is nearly complete. Then, the process to achieve 100% of completeness is slower. This learning speed could still be improved. Today, Kapuer starts without any initialization and has to learn users preferences from scratch. With an initialization on these preferences, the number of requests needed to propose the first rule could be reduced.

VI. Conclusion and Future Work

We have presented in this article a tool for permission management on Android. Unlike other approaches, Kapuer does not only provide a way to modify what permissions an application can use. It learns from users' behavior to help them and advise them by proposing rules with different levels of abstractions. This way, they can protect their privacy more easily, without needing knowledge about access control models or policy's structure. Our evaluations show that hundreds of permissions can be handled with a limited number of actions by using abstractions. When Android 6.0 sacrifices control of privacy for the sake of simplicity with concepts of protection levels and groups of permissions, Kapuer learns and proposes both fine-grained and abstract privacy rules. In addition, Kapuer supplies users with features to control the level of abstraction of privacy rules.

The current version of Kapuer runs on Android 4.4. For short term future works, we will upgrade it to Android 6.0 in order to benefit from new features introduced by this system. For the moment, each time a request is denied, Kapuer makes Android act as if the application does not have the permission. Since developers of Android 4.4 applications do not manage this case, some applications crashed. This issue will be resolved on Android 6.0 because now developers shall handle permission verification before trying to use it. We will also take advantage of the new Android permission request interception to implement our interactions with users. Finally, we will integrate new Android 6.0 information (protection levels and groups of permission) as new meta-criteria. As a consequence, Kapuer will be easier to maintain.

One of the initial goal when we designed Kapuer was to inform people about privacy risks. For longer term research, we want to go further in that direction and not only inform people but also educate them about privacy issues. As an example, explain them the consequences of granting some permissions to an application. The more people understand these risks, the better their privacy decisions will be.

Finally, Kapuer learns users preferences from scratch. A large number of requests is needed before any proposition can be made to the user. It is possible to improve the beginning of the learning phase by initializing the system. We are currently making surveys with different kind of users to find the best way to initialize these users' preferences.

References

[1] Assemblée Générale Des Nations Unies, "Déclaration universelle des droits de l'homme," *Résolution 217A (III)*, vol. 10, 1948.

[2] A. Oglaza, P. Zarate, and R. Laborde, "Kapuer: A decision support system for privacy policies specification," *Annals of Data Science*, vol. 1, no. 3-4, pp. 369–391, 2014. [Online]. Available: http://dx.doi.org/10.1007/s40745-014-0027-3

[3] S. Barker, "The next 700 access control models or a unifying meta-model?" in *Proceedings of the 14th ACM symposium on Access control models and technologies*. ACM, 2009, pp. 187–196.

[4] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The nist model for role-based access control: towards a unified standard," in *ACM workshop on Role-based access control*, vol. 2000, 2000.

[5] F. Cuppens and A. Miège, "Modelling contexts in the or-bac model," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*. IEEE, 2003, pp. 416–425.

[6] J.-W. Byun, E. Bertino, and N. Li, "Purpose based access control of complex data for privacy protection," in *Proceedings of the tenth ACM symposium on Access control models and technologies*. ACM, 2005, pp. 102–110.

[7] Q. Ni, E. Bertino, J. Lobo, C. Brodie, C.-M. Karat, J. Karat, and A. Trombeta, "Privacy-aware role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 3, p. 24, 2010.

[8] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Crêpe: A system for enforcing fine-grained context-related policies on android," *Information Forensics and Security, IEEE Transactions on*, vol. 7, no. 5, pp. 1426–1438, Oct 2012.

[9] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes, "Moses: Supporting and enforcing security profiles on smartphones," *Dependable and Secure Computing, IEEE Transactions on*, vol. 11, no. 3, pp. 211–223, May 2014.

[10] oasis xacml committee. extensible access control markup language (xacml) version 3.0. [Online]. Available: https://www.oasis-open.org/committees/xacml/

[11] V. Arena, V. Catania, G. La Torre, S. Monteleone, and F. Ricciato, "Securedroid: An android security framework extension for context-aware policy enforcement," in *Privacy and Security in Mobile Systems (PRISMS), 2013 International Conference on*, June 2013, pp. 1–8.

[12] B. Stepien, A. Felty, and S. Matwin, "A non-technical xacml target editor for dynamic access control systems," in *Collaboration Technologies and Systems (CTS), 2014 International Conference on*. IEEE, 2014, pp. 150–157.

[13] A. Oglaza, R. Laborde, and P. Zarate, "Authorization policies: Using decision support system for context-aware protection of user's private data," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, July 2013, pp. 1639–1644.

[14] P. G. Keen and M. S. S. Morton, *Decision support systems: an organizational perspective*. Addison-Wesley Reading, MA, 1978, vol. 197, no. 8.

[15] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 6, pp. 734–749, 2005.

[16] A. J. Jeckmans, M. Beye, Z. Erkin, P. Hartel, R. L. Lagendijk, and Q. Tang, "Privacy in recommender systems," in *Social media retrieval*. Springer, 2013, pp. 263–281.

[17] A. Friedman, B. P. Knijnenburg, K. Vanhecke, L. Martens, and S. Berkovsky, "Privacy aspects of recommender systems," in *Recommender Systems Handbook*. Springer, 2015, pp. 649–688.

[18] B. Rashidi, C. Fung, and T. Vu, "Dude, ask the experts!: Android resource access permission recommendation with recdroid," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE, 2015, pp. 296–304.

[19] A. Oglaza, P. Zaraté, and R. Laborde, "KAPUER: A Decision Support System for Protecting Privacy (regular paper)," in *Group Decision and Negotiation (GDN), Toulouse, France, 10/06/2014-13/06/2014*, ser. LNBIP, P. Zaraté, G. Kersten, and J. Hernandez, Eds., no. 180. http://www.springerlink.com: Springer, juin 2014, pp. 100–107. [Online]. Available: http://oatao.univ-toulouse.fr/13069/