User-centric Android Flexible Permissions

Gian Luca Scoccia*, Ivano Malavolta†, Marco Autili‡, Amleto Di Salle‡ and Paola Inverardi‡
*Gran Sasso Science Institute, L'Aquila, Italy - gianluca.scoccia@gssi.infn.it
†Vrije Universiteit Amsterdam, Amsterdam, The Netherlands - i.malavolta@vu.nl
‡University of L'Aquila, L'Aquila, Italy - {marco.autili,amleto.disalle,paola.inverardi}@univaq.it

***Abstract*—Privacy in mobile apps is a fundamental aspect to be considered, particularly with regard to meeting end user expectations. Due to the rigidities of the Android permission model, desirable trade-offs are not allowed. End users are confined into a secondary role, having the only option of choosing between either privacy or functionalities. This works proposes a user-centric approach to the flexible management of Android permissions that empowers end users to specify the desired level of permissions on a per-feature basis.**

***Keywords*-Mobile Apps, Android Permissions, Privacy.**

I. Problem Statement

The trustability of mobile apps is one of the main factors to be considered, given the constant need of these apps to access sensitive and private information of end users [2], to the point that consumers are willing to pay a premium for privacy protection [5]. The current Android permissions model suffers of a number of rigidities related to, e.g., the granularity level of the permissions, the timing at which permissions are granted, the fact that the permissions model considers all users as equal, etc.

As a result, users have the only option of either not installing the app or granting all permissions, often by relying only on incomplete information about possible risks. This lack of flexibility may have a negative impact on the success of a mobile app. End users may decide to not use the app rather than granting permissions they feel uneasy about. Indeed, this issue was partially considered in the design of release 6 of the Android. However, many problems remain unsolved. That is, Android permissions and their usage still attract a considerable research interest. The outcome is that Android users grant permissions with a minimal comprehension of what the implications of their decisions will be [2], without having the possibility of balancing between functionalities and privacy.

II. Proposed Solution

This work proposes Android Flexible Permissions (AFP), a more *user-centric approach to flexible permissions management*. End users are allowed to specify and customize fine-grained permission levels according to their own subjective privacy concerns. AFP leverages a novel permission model through which app permissions are specified on a per-feature basis. Differently from the current Android permission model, AFP empowers end users to selectively grant permission by specifying (i) the desired *permission levels* (e.g., access to the contacts list can be granted to all contacts that do not belong to specific circles of people like relatives or close friends), and (ii) the *features* of the app in which the specified permission levels are granted (e.g., access to the relatives circle in the contacts list can be granted only during a video call in a messaging app). AFP offers a dedicated external mobile app for that purpose.

From the developers point of view, the goal is to provide a means to create apps that *dynamically adapt to user-defined permission levels* with a limited additional effort. Developers create their mobile apps as usually, without using any additional library or tool. In order to comply with AFP, a developer is provided with automatic support by the AFP Web application that (by means of a wizard) allows to (i) define the features offered by the mobile app, and (ii) map each feature to the components that implements it, i.e., Android activities, services, broadcast receivers, or content providers. Features and mappings are then used to automatically retrofit the app so that it will be able to dynamically handle fine-grained permission levels at runtime.

III. The AFP Approach

AFP involves the following main components:
– *AFP App*, an app from which users can manage their own flexible permissions;
– *AFP Library*, a library to enforce permissions at runtime;
– *AFP Server*, a web app that allows developers to automatically retrofit an existing app so as to comply with AFP. It also offers mechanisms for signing and verifying AFP-enabled apps.

With reference to Figure 1, in the following the AFP workflow is described from the developer point of view, and from the end-user point of view.

App developer perspective – When an app X is ready to be published (right-hand side of Figure 1), the developer can send the APK archive of X to the *AFP Server* so to enable AFP (1). Internal to the server, all the Android components of X are extracted, i.e., its constituent activities, services, broadcast receivers, and content providers. Then, the developer use a *web-based editor* for (i) defining the features of X in terms of their name and description (later used by end users), and (ii) mapping each one of them to (a subset of) the extracted Android components implementing

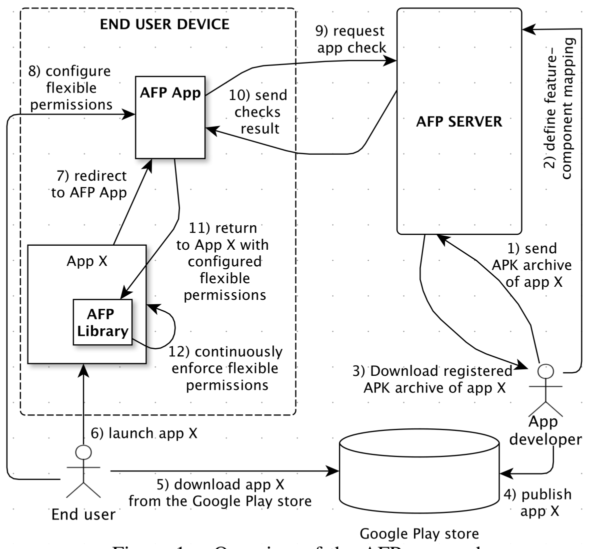
END USER DEVICE
8) configure flexible permissions
AFP App
10) send checks result
9) request app check
AFP SERVER
2) define feature-component mapping
7) redirect to AFP App
11) return to App X with configured flexible permissions
App X
AFP Library
12) continuously enforce flexible permissions
1) send APK archive of app X
3) Download registered APK archive of app X
App developer
6) launch app X
End user
5) download app X from the Google Play store
4) publish app X
Google Play store

Figure 1. Overview of the AFP approach

it (2). This step is the only additional effort we request to developers, and it is heavily guided by the editor together with the automatic extraction of Android components. The output is a *feature-component mapping model*, specifying the mapping between app features and Android components.

App X is then statically analyzed and automatically retrofited so as to enable AFP on it. For this purpose, the following operations are performed (those are totally transparent to the developer): (i) automatically includes our *AFP Library* in the app; (ii) instruments X so that all calls to sensitive Android APIs are proxified and redirected to the *AFP Library*; (iii) injects the code in the main activity of X for allowing the end user to switch to the *AFP App* when launching X for the first time; (iv) assigns a unique secret key to the app X, which will be used at runtime by the *AFP checker*; (v) creates a new record into a repository of registered apps within the server; (vi) rebuilds and re-sign X as a new APK archive. Finally, the instrumented APK of X is made available to the developer (3), who can then proceed with the publication in the Google Play Store (4).

End user perspective – The user workflow has been designed to be as much as possible seamless and easy to follow, with the objective of minimizing the effort required to end-users to specify flexible permissions. Users can download and install (5) apps that adopt the AFP system directly from the Google Play Store since no modifications to the Android OS are required. Upon the first launch of the newly installed app (6) they are redirected to the *AFP App* (7), which in turn invites them to configure the permissions.

Once inside the *AFP App*, the user can specify preferences for each permission requested by the app (8). In the meanwhile, the *AFP App* also interacts with the *AFP Server* (9) in background. The server uses an internally generated secret key to check the app installation and verify the developer's identity, hence certifying that nobody tampered with the *AFP Library*. Moreover, it verifies that the APK downloaded from the Google Play Store is exactly the same as the one produced by our approach. When the results of the checks are ready (10), and the configuration phase has finished, the user will be automatically redirected back to the newly installed app, together with the now configured flexible permission model (11). The permission model is now associated with the AFP-enabled app and the user can continue with normal app usage, in a completely transparent way, i.e., no further user interaction or dialogs are required.

Whenever needed, the access to private or sensitive resources will be granted by the *AFP Library* according to the specified permissions (12). Intuitively, the *AFP Library* proxifies each call of the app to sensitive Android APIs (e.g., call to the Android geolocation manager), hence wrapping the access to sensitive resources. Should the permission model allow the requested access, the execution proceeds normally; otherwise, functionalities are degraded by *mocking* the requested data according to the permission levels specified by the end user (this aspect is inspired by the Mockdroid approach by Beresford et al. [1]). For example, if the end user allows only city-level geolocation, when the app calls the Android location manager, the *AFP Library* intercepts that call and returns the geographical center of the city where the user is, instead of her precise location.

The *AFP App* also allows to specify default levels for the permissions (e.g., geolocation is allowed only at the city-level, independently of the app requesting it), that will be used as a basis during the configuration of the flexible permissions for any newly installed AFP-compliant app. This characteristic permits to speed up the configuration of the permissions for each newly installed AFP-enabled app.

IV. Future Work

Much needs to be still done. In the short term, our primary goal is to finalize the implementation of AFP by using a combination of Java and Web technologies. Then, a thorough empirical evaluation of the approach will follow by leveraging aptly defined experiments on a dataset created in the context of a previous research in which we mined the top 500 most popular free apps for each category of the Google Play Store [3], [4]. In the mid term, the goal is to define a procedure for (semi) automatically extracting the features provided by an Android app from its binary or source code. Moreover, as most of the approaches in the literature using static analysis, the use of reflection, self-decrypting code, or obfuscation techniques in general challenge our approach. We will investigate the possibility of realizing a hybrid approach combining static analysis with dynamic flow analysis. We believe that such a hybrid approach may result in a valid and viable compromise towards mitigating this challenge.

References

[1] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54. ACM, 2011.

[2] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, pages 3:1–3:14, 2012.

[3] I. Malavolta, S. Ruberto, T. Soru, and V. Terragni. End users perception of hybrid mobile apps in the google play store. In *Proceedings of the 4th International Conference on Mobile Services (MS 2015)*. Institute of Electrical and Electronics Engineers (IEEE), 2015.

[4] I. Malavolta, S. Ruberto, T. Soru, and V. Terragni. Hybrid mobile apps in the google play store: An exploratory investigation. In *Proceedings of the 2nd International Conference on Mobile Software Engineering and Systems (MobileSoft 2015)*, pages 56–59. Institute of Electrical and Electronics Engineers (IEEE), 2015.

[5] A. P. F. Serge Egelman and D. Wagner. Choice architecture and smartphone privacy: There's a price for that. In *The Economics of Information Security and Privacy*, pages 211–236. 2013.