Enhancing Trustability of Android Applications via User-Centric Flexible Permissions

Gian Luca Scoccia, Ivano Malavolta, Marco Autili, Amleto Di Salle, Paola Inverardi

Abstract—The Android OS market is experiencing a growing share globally. It is becoming the mobile platform of choice for an increasing number of users. People rely on Android mobile devices for surfing the web, purchasing products, or to be part of a social network. The large amount of personal information that is exchanged makes privacy an important concern. As a result, the trustability of mobile apps is a fundamental aspect to be considered, particularly with regard to meeting the expectations of end users. The rigidities of the Android permission model confine end users into a secondary role, offering the only option of choosing between either privacy or functionalities. In this paper, we aim at improving the trustability of Android apps by proposing a user-centric approach to the flexible management of Android permissions.The proposed approach empowers end users to selectively grant permission by specifying (i) the desired level of permissions granularity and (ii) the specific features of the app in which the chosen permission levels are granted. Four experiments have been designed, conducted, and reported for evaluating it. The experiments consider performance, usability, and acceptance from both the end user's and developer's perspective. Results confirm confidence on the approach.

Index Terms—Android Permissions, Static Analysis, Trustability.

1 Introduction

The Android market is experiencing an explosive growth in the last years. It is the smartphone OS market leader among all age segments in the US, UK, and other countries [1]. People rely on Android mobile devices for surfing the web, purchasing products, or to be part of a social network. Represented by the well known Google Play Store, the Android apps market now counts more than two millions apps, downloaded billions of times per year [2].

In this landscape, the Android permission model plays a key role, given the constant need of mobile apps to access sensitive and private information of end users [3], [4], [5], to the point that consumers have to choose between lower prices and more privacy protection [6], [7]. The Android permission model can influence the success of mobile apps as well. Indeed, a recent empirical study found that privacy-aware users with negative concerns on an app's runtime permissions tend to give negative reviews to the whole app, thus confirming the importance of how permissions are managed for the success of Android apps [8].

The current Android permission model suffers of a number of rigidities related to, notably, the granularity level of the permissions, the timing at which permissions are granted, the fact that the permission model considers all users as equal [9], [10]. On the one hand, this lack of flexibility may have a negative impact on the success of a mobile app; on the other hand, it neglects the rights of the user to fully express her desires and exert her control on how and by whom her data are used. According to a survey conducted among adult Americans [11], 91% of participants believe that consumers have lost control over how personal information is collected and used by companies, and that most participants would like to do more to protect their personal information.

In this paper, we propose Android Flexible Permissions (AFP), a *user-centric approach to flexible permissions management* aimed at empowering end users to play an active role with respect to Android permissions. AFP embraces the European vision of next generation internet, more human-centric and concerned with privacy protection by giving control back to users [12]. End users are allowed to specify and customize fine-grained permission levels on private or sensitive resources, according to their own subjective privacy concerns, risk taking attitudes and sense of trust. AFP leverages a novel permission model through which fine-grained app permissions are specified on a *per-feature*[1] basis. Differently from the current Android permission model, AFP empowers end users to selectively grant finer-grained permissions by specifying (i) the desired *permission levels*[2] (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 the usage of the video call feature in a messaging app). AFP offers a dedicated external mobile app for managing flexible permissions.

From the developer's point of view, AFP enables apps to support *user-defined permission levels* with very limited

- *I. Malavolta is with the Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands. E-mail: i.malavolta@vu.nl*
- *G. L. Scoccia, M. Autili, A. Di Salle, and P. Inverardi are with the Department of Information Engineering, Computer Science and Mathematics (DISIM), University of L'Aquila, Italy. E-mail: {gianluca.scoccia, marco.autili, amleto.disalle, paola.inverardi}@univaq.it*

Manuscript received Month XX, 20XX; revised Month XX, 20XX.

1. Building on the definition provided in the Software Engineering Body of Knowledge [13], the term feature is used to denote a functionality offered by the app to the user.

2. The granularity at which a sensible resource/data can be accessed.

additional effort. Developers can work on their mobile apps as usual, without using any additional library or tool. In order to use AFP, a developer is provided with automatic support by the AFP Web application that (by means of a guided workflow) allows to (i) define the features offered by the mobile app; (ii) map each feature to the components that implement it, i.e., Android activities, services, broadcast receivers, or content providers. Given the feature-component(s) mappings, AFP leverages static control-flow analysis for *automatically* retrofitting the app so to make it able to handle fine-grained and feature-based permission levels.

We evaluated AFP by designing, conducting, and reporting four independent experiments aimed at empirically investigating on key aspects of AFP. Specifically, we assessed the performance of the AFP instrumenter via 1,277 real-world apps, the performance at runtime of 7 AFP-enabled real-world apps, the usability and acceptance of AFP for both end users and developers (involving 47 and 11 subjects, respectively).

The main **contributions** of this paper are:

1) an in-depth discussion of the characteristics and challenges of the Android permission model;
2) the definition of a new *flexible permission model* for Android apps;
3) the definition of an *approach* that empowers end users to specify and enact flexible permissions for Android apps;
4) a *publicly available implementation* of AFP using a combination of Java and Web technologies;
5) the results of the *empirical evaluation* of AFP, and a complete *replication package* for independently assessing and replicating the aforementioned experiments.

An initial version of this work appeared as a poster paper at the 2017 ACM/IEEE International Conference on Software Engineering [14]. The new contributions of this journal version include: the precise definition of the AFP permissions data model, the app features specification and mapping mechanism, the algorithm for automatically instrumenting Android apps in order to make them AFP-enabled, and its implementation and technological aspects. Another completely new contribution of this paper is the empirical evaluation of AFP, for which we designed, conducted, and reported 4 independent experiments about its performance, usability, and acceptance from the perspective of both end users and developers. This paper also includes a thorough discussion of related work.

The remainder of the paper is structured as follows. Section 2 discusses in details the problem we want to solve and motivates our research. Section 3 presents the AFP approach, and Section 4 describes its implementation details. Section 5 reports on the experimental evaluation we performed and discusses the limitations of AFP. Section 7 presents related work, and Section 8 closes the paper and discusses future work.

2 The need of a new perspective

In this section, we first analyze the evolution of the Android permission system by discussing the issues of both the older *install-time permission system* (Section 2.1) and the most recent *usage-time permission system* used by the latest Android releases (Section 2.2). Then, in Section 2.3, we discuss the need of a new perspective according to which users must be given the possibility of specifying their own rules on how sensitive resources should be accessed by apps.

2.1 Install-time permission system

Traditionally, up to version 5.1.1 (i.e., Android API level 22), Android makes use of an install-time permission system to regulate the access to sensible APIs of the platform. Developers have to declare upfront if their apps require access to security- and privacy-relevant parts of the platform. Very little control is in the hands of the end user, who can only decide to grant or reject all permissions to access such parts of the Android APIs before app installation, and can only rely on warning dialogs to assess possible risks (often using extremely broad wording). If the user only agrees with a subset of the permissions, she has to abort the app installation. This amounts to force the end user to either grant all permissions or not install the app. It has been observed that users routinely decide to not install an app because of the permissions it requires [3]. Moreover, usability studies show that only a minority of users have a reasonable comprehension of warning dialogs [3], [15], [16]. Dialogs have been found to be vague and devoid of context, as users have no way to know what app features the install-time permissions correspond to.

2.2 Usage-time permissions system

Starting with Android 6 (i.e., Android API level 23), access to privacy- and security-relevant parts of the platform can be enforced by a usage-time permission system. While the app is running, the system checks whether the app functionality that is going to be used for the first time has the required permissions or not. Users are prompted for confirmation when the functionality attempts to access a restricted part of the platform.

While runtime permissions provide more detail over the specific functionalities of the app affected by the permissions, hence helping users in making their decision, permissions are still granted on a whole-app basis, i.e., *once granted the permission is valid for the entire app*, further reducing control on how and when private data is accessed. Users can potentially revoke permissions already granted to an app but this requires exiting the app, accessing their device system settings and manually changing the permission settings of the app. The whole process is unwieldy, and users are unlikely to do so [17].

In addition, in the Android platform, permissions are grouped into *permission groups*. A permission group is a set of two or more permissions that reference the same resource. For instance, the permissions READ_CONTACTS and WRITE_CONTACTS both belong to the CONTACTS permission group. Whenever an app tries to access a protected resource, users are prompted for confirmation only if no other permission in the belonging permission group is already granted. This is done in order to minimize a phenomenon known as *warning fatigue*, which has been

observed in the past in other usage-time permissions systems [18], [19], [20] – i.e., users may become insensitive to warning messages after being exposed to an excessive amount. While the concept of permission group can be effective in reducing warning fatigue, it also makes the permissions more coarse grained. Multiple permissions are in fact granted with a single confirmation, further reducing the user control on the resources that will be accessed with respect to the granted permissions.

2.3 A new perspective and its challenges

The issues discussed above can be addressed by reconsidering the way permissions are declared, requested and enforced, shifting to a process where users are given the means to establish their own rules on how sensitive resources are accessed, thus aiding them in retaining control over their data and consequently increasing their trust on the platform.

We need to acknowledge that end users can have diverse privacy desiderata and concerns [21]. As a consequence, we have to move away from a one-size-fits-all approach in favor of more flexible solutions. Under this perspective, the goal of AFP is to provide users with the means to specify permission preferences in a more fine-grained and customizable manner, leaning on finer-grained representations of both app functionalities and sensible resources.

On the other side of the coin, developers should be provided with the means to create, with limited additional effort, apps that adapt to user preferences (possibly changing on the fly), coupled with new solutions for acquiring users preferences that avoid warning fatigue [3].

As already introduced, we denote a functionality offered by the app to the user with the term *feature*; the granularity at which a resource can be accessed with the term *level*. Examples of features include: showing the list of restaurants nearby, sending a message to a friend, creating a new post in a social networking app, etc. Considering user's location as a resource, examples of levels are: full access to it, precision limited to current city, precision limited to current region, etc.

3 The AFP Approach

Android Flexible Permissions (AFP) grants permissions on a per-feature basis by (i) keeping track of user privacy preferences, and (ii) automatically enacting and enforcing them at runtime. AFP is composed of the following main components:

- *AFP App*, an app from which users can manage their own flexible permissions;
- *AFP Library*, a library for access control at runtime;
- *AFP Server*, a web app that allows developers to automatically retrofit an existing app in order to enable AFP in it. It also offers mechanisms for signing and verifying AFP-enabled apps.

With reference to Figure 1, in the following we describe the workflow of AFP. Section 3.1 describes the steps to be followed by developers before publication, in order to make their apps compatible with AFP. Section 3.2 describes the steps to be followed by users upon the first app execution, in order to specify the desired permissions.

3.1 App developer perspective

The developer workflow is designed to minimize the effort needed to create apps compatible with the flexible permission system.

Developers create their mobile apps as usual, without using any additional library or tool. 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). As detailed in Section 3.5, the *Android Components Extractor* extracts all the Android components of X, i.e., its constituent activities, services, broadcast receivers, and content providers (2). Then, the developer uses 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 it (3). Step 3 is the only additional effort we request to developers, and it is greatly facilitated by the web-based editor together with the automatic extraction of Android components. The output of this phase is the *feature-component mapping model*, specifying the mapping between app features and Android components.

The *AFP Instrumenter* statically analyzes and automatically retrofits app X to enable AFP on it (4). The instrumenter performs the following operations (that 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 App Checker* (12); (v) creates a new record into the repository of registered apps; (vi) rebuilds and re-sign X as a new APK archive. Finally, the instrumented APK of X is made available to the developer (5), who can then proceed with the publication of the APK in the Google Play Store (6).

3.2 End-user perspective

The user workflow is designed to minimize the end user effort to specify flexible permissions.

Users can download and install (7) 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 (8) they are redirected to the *AFP App* (9), which in turn invites them to configure the flexible permissions. Should the user be unwilling to do so, she can immediately abort the process, and the app usage will continue with the standard permission system provided by the Android platform. The same happens if, during the configuration process, the *AFP App* is not found installed on the user's device.

Once inside the *AFP App*, for each feature of the app the user can specify her own permission preferences (10). This aspect of AFP allows to address a well-known problem in the current Android permission model, i.e., the fact that users tend to not understand why they are being asked for certain permissions, often complaining about this aspect in their reviews in the Google Play store [8]. Indeed, by explicitly asking end users to define the permission levels on a per-feature basis allows users to (i) read the description

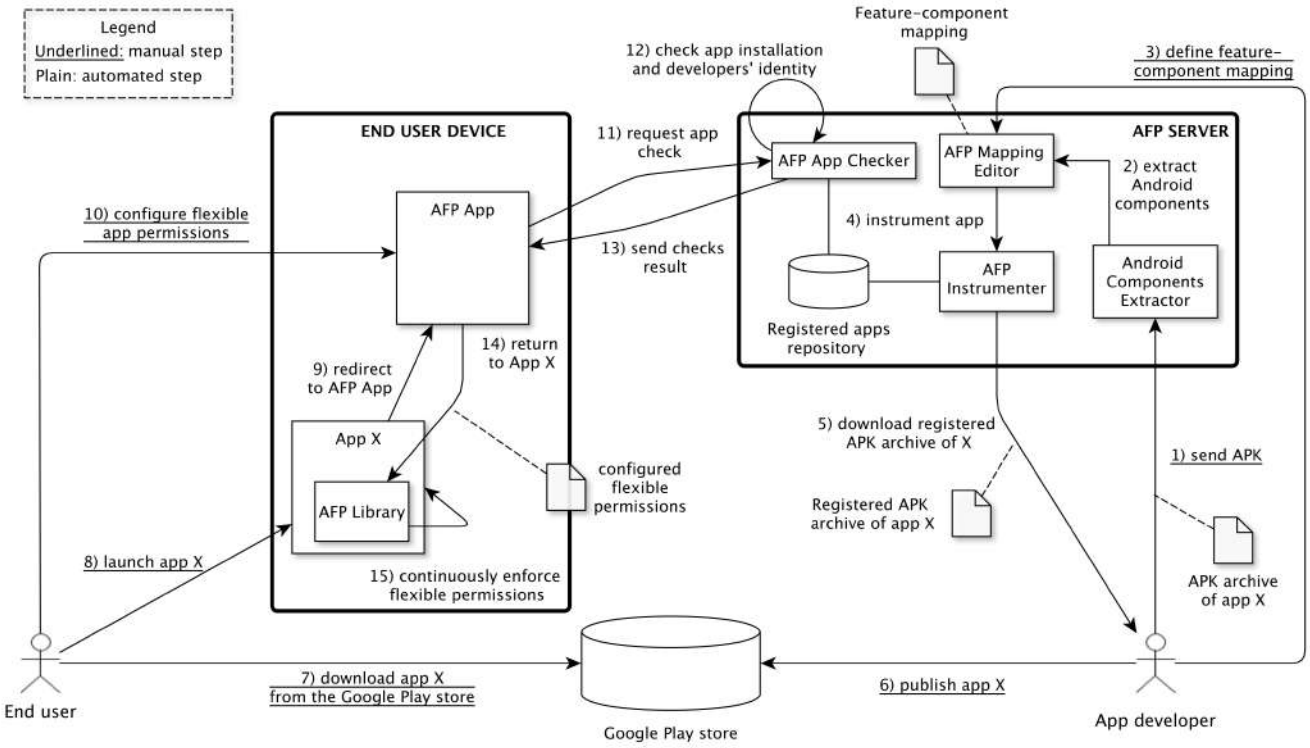


Fig. 1. Overview of the AFP approach (underlined labels represent manual steps, whereas all the others are fully automatic)

of each feature provided by the developer and (ii) better understand why certain permissions are being requested in the context of each specific feature of the app, rather than within the app as a whole. By using AFP, end users have a more transparent view of the features provided by their apps, and a better knowledge about the context in which (sensitive) permissions are requested by the app.

While the end user is setting her desired permissions, in background, the *AFP App* also interacts with the *AFP Server* (11). The *AFP App* and the *AFP Server* communicate via the HTTPS protocol. The server also uses an internally generated secret key (4) 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 one produced by our approach (12). When the results of the checks are ready (13), and the configuration phase finishes, the user will be redirected to the newly installed app, together with the configured flexible permissions configuration (14). The permissions configuration is then associated with the AFP-enabled app and the user can continue with app usage, in a completely transparent way, i.e., no further user interaction or dialogs are required.

The access to private or sensitive resources will be granted by the *AFP Library* according to the specified permissions configuration (15). 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.

The *AFP App* 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.

3.3 Flexible permission data model

This section presents the AFP data model to which permission configurations conform (Figure 2). It is based on the following core concepts:

- *Resource* represents a sensitive part of the Android platform whose access can be controlled by the AFP library. In AFP, resources are both physical parts of the device, such as the device camera and microphone, and logical ones, such as the user's contacts book.
- *Feature* represents a user-level functionality of the app. Every *Feature uses* one or more *Resources*. In addition, every *Feature* is directly connected through the *realizedBy* relation with one or more *AndroidComponent* instances, each of them representing one or more source code files inside the app.
- A *PolicyItem* regulates access to the *Resources* used by the *Feature*. It represents a single access restriction rule that can be imposed upon one or more *Resources*. For instance, a *PolicyItem* could specify that the access to the device camera has to be forbidden, or that only the user's city should be shared when the device is queried for the user position.
- *AccessPolicy* is a conjunction of one or more *PolicyItems* and it is linked to a *Feature*.

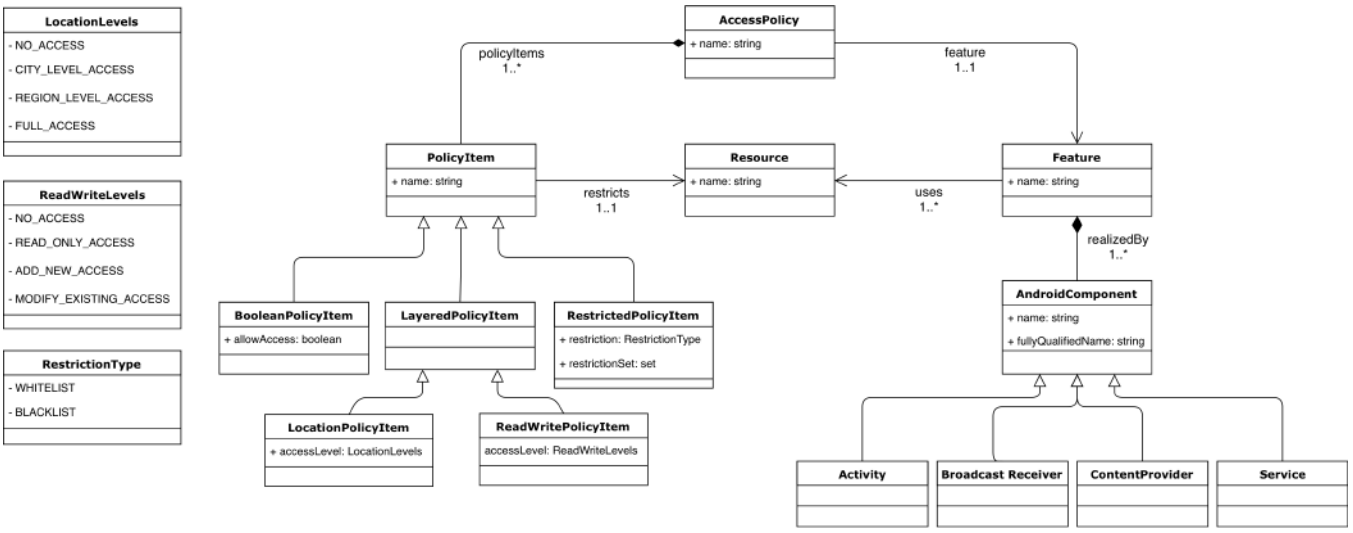


Fig. 2. Flexible permission data model

The remaining classes in the data model define the restrictions that can be enforced:

- *BooleanPolicyItem* permits either full access to the resource or no access at all.
- *RestrictedPolicyItem* restricts the access to a *restrictionSet*, whose type can be either *BLACKLIST* or *WHITELIST*. For instance, it can be used to restrict access to the contacts list to only contacts that do not belong to specific circles of people like relatives or close friends.
- *ReadWritePolicyItem* and *LocationPolicyItem* extend the *LayeredPolicyItem* class and allow for granting access to a *Resource* at incremental levels. Higher levels are used for less restrained accesses to the *Resource*. *ReadWritePolicyItem* has four possible levels. In particular, *ADD_NEW_ACCESS* and *MODIFY_EXISTING_ACCESS* allow for adding new records to the *Resource* and editing existing ones, respectively. For example, it is possible to grant access to the sms messages but prevent creating and sending new ones. *LocationPolicyItem* instead provides four levels of different precisions for access to the user position.

3.4 Features specification

The AFP approach involves (i) the automatic extraction of Android components composing the mobile app (step 2 in Figure 1) and (ii) the definition of a mapping between features and the Android components implementing them in the app (step 3).

For what concerns step 2, all relevant information is extracted from both the XML file of the Android manifest and the bytecode of the Java classes of the app. The output of this step is a fragment of configuration conforming to the flexible permission data model described in Section 3.3 containing only instances of the AndroidComponent class and its subclasses.

In order to specify the mappings (step 3), the developer uses a form (Figure 3) where she can declare the main features of the app and, by means of check boxes, associate them to the automatically extracted Android components. The final result of this step is the complete configuration, which also includes the required instances of the Feature class.

Fig. 3. Feature to Android components mapping specification form

3.5 App instrumentation

The AFP approach involves also the automatic instrumentation of the app (step 4 in Figure 1). Such step is carried on by the *AFP Instrumenter* that performs a set of operations that can be grouped into three main phases: (*i*) decomposing the input APK, (*ii*) rewriting the app bytecode, and (*iii*) repackaging the rewritten app.

The goal of the first phase is to extract, from the compiled binary files of the app under analysis, the app Java bytecode. Although the bytecode is a low-level representation, it is suitable to perform analysis and instrumentation. In turn, the third phase performs the reverse operation, transforming the instrumented bytecode back into a compiled binary file. The logic behind the first and the third phases is straightforward, and they are both carried out using freely available tools arranged in a pipe-and-filter pattern. The adopted tools are described in Section 4.

```
   input : App, an Android app
           C, set of Android components ∈ App
           F, set of features ∈ App
           M, mapping of elements of F to C
   output: App', AFP-compliant version of App

1  begin
2      foreach c ∈ C do
3          scan C to extract SC, set of sensitive calls ∈ C
4          if SC ≠ ∅ and c ∉ M then
               /* found sensitive call not mapped to any
                  feature, raise error and terminate */
5              break
6          end
7          if c is App main activity then
8              add intent trigger towards AFP app to
               allow for permissions configuration
9          end
10         foreach sc ∈ SC do
11             identify r, resource affected by sc
12             foreach f ∈ F do
13                 if c is mapped to f then
14                     add r to f_r, set of resources used by
                       f
15                 end
16             end
17             replace sc with sc', call to AFP library
               wrapper
18         end
19     end
20 end
```

Algorithm 1: Rewriting algorithm of AFP instrumenter

During the second phase, rewriting of the app bytecode is performed by an ad-hoc analyzer, following Algorithm 1. Given as input an Android app and its feature-component mappings, the algorithm returns an AFP-compliant version of the app. In order to do so, the algorithm iterates over all Android components that constitute the app (line 2 in Algorithm 1) and extracts from each of them the set of sensitive API calls performed inside its logic (line 3). Each call is analyzed for the purpose of identifying the affected resource (lines 10-16), and creating and binding instances of the Resource class in the AFP data model. Each sensitive call is then replaced with a corresponding call to the AFP library (line 17), which contains a proxy class for each of the Android APIs that enable access to restricted parts of the platform (further details in Section 3.6). While performing the rewriting, the algorithm also checks that all classes containing calls to restricted parts of the Android API belong to a feature (lines 4-6). This ensures that the developer does not leave some Android components out of the mapping. Additional code (via Android *Intents*) is added in the main activity (i.e., the app entry point) to set up communication between the app under rewriting and the *AFP App*. It enables the configuration of the flexible permissions on the startup of the app (lines 7-9).

The illustrative example, given in Figure 4, shows bytecode rewriting of an API call for reading the last registered user's location. In the original version (Listing A), a LocationManager object is loaded from a local variable and pushed onto the stack (instruction 1). Then, the constant string "network", used as a parameter in the upcoming method invocation, is also pushed onto the stack (instruction 2). Both are consumed from the stack with an invocation to the virtual method getLastKnownLocation (instruction 9). Finally, the Location object resulting from the method call is read from the stack and stored in a static field (instruction 10). In the rewritten version (Listing B), the invocation to the virtual method getLastKnowLocation has been replaced with a static call to its proxified method in the *AFP Library* (instruction 9). Since this method also requires, as an array parameter, the identifiers of features to which it belongs, this information is also pushed onto the stack (instructions 4-8).

3.6 Permissions enactment and enforcement

As described in Section 3 (see Figure 1), a user can download and install AFP-enabled apps from the Google Play Store, as she would normally do for all apps. Upon starting a newly installed app for the first time, she is redirected to the *AFP App*, which enables permission enactment by allowing her to configure the flexible permissions associated to it. Two screenshots of the app are presented in Figure 5. During the configuration, she is presented the list of features offered by the AFP-enabled app. For each feature, the accessed sensitive resources are listed and, for each one of them, she can set her preferences, hence regulating the access to the resources for that single feature. As an example, consider a user interested in the Facebook app. After downloading and installing it on her device, on the first run she is presented with the list of app features, i.e., *Wall*, *Messaging*, *Events*, etc. Assuming that she does not want her friends to know her exact location every time she posts on her Wall, she can restrict the precision of the Location resource for the Wall feature, while leaving it unchanged for Events in order to still discover ongoing events nearby.

While the configuration is ongoing, the *AFP App* establishes communication with the *AFP Server* and checks the validity of the app secret key, promptly raising a warning should it be different from the one stored on the server. If there is no Internet connection and the validity of the app secret key cannot be checked, *AFP App* raises a warning and makes the user aware of the potentially dangerous situation. The warning is not blocking and, if the user agrees to proceed, it does not prevent the usage of the app. It is important to note that this specific situation occurs only once during the whole lifetime of each AFP-enabled app since the check of the identity of its developer is performed at its very first launch after the installation.

The configuration procedure can be terminated at any time and, upon termination, the configured permissions configuration is transmitted back to the calling app, and stored by the *AFP Library*.

Access control is in the hands of the *AFP Library*, which contains a proxy class for each of the Android APIs that enable access to restricted parts of the platform. At runtime, whenever invoked, the methods contained in the proxy classes perform a check against the configured permission

```
1   aload_0
2   ldc network
3
4
5
6
7
8
9   invokevirtual android/location/LocationManager.getLastKnownLocation:(
            Ljava/lang/String;)Landroid/location/Location;
10  putstatic location:Landroid/location/Location;
```

Listing A

```
1   aload_0
2   ldc network
3   iconst_1
4   anewarray java/lang/String
5   dup
6   iconst_0
7   ldc WeatherForecast
8   aastore
9   invokestatic gssi/aq/it/afplibrary/AFPLocationManager.
            getLastKnownLocation:(Landroid/location/LocationManager;Ljava/lang/
            String;[Ljava/lang/String;)Landroid/location/Location;
10  putstatic location:Landroid/location/Location;
```

Listing B

Fig. 4. Comparison between an original byte code file (Listing A) and the rewritten version produced by the *AFPInstrumenter* (Listing B).

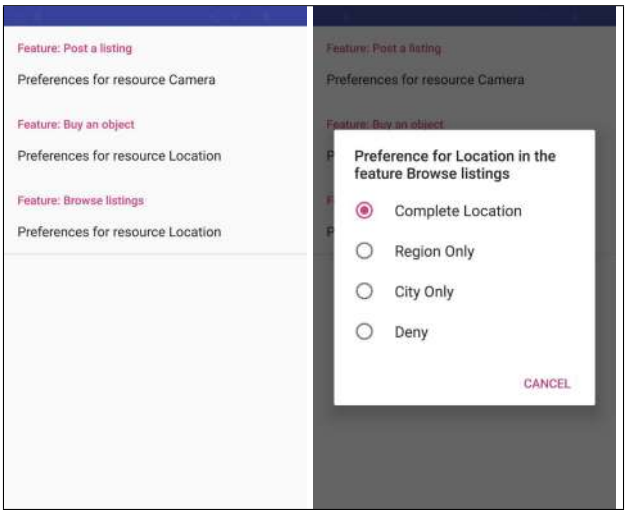


Fig. 5. Two screenshots of the AFPApp: list of features presented to the end user and possible levels for Location

model and allow access to the restricted parts of the platform only if admitted by the model. If access to a resource has to be allowed only at certain level, then the *AFP Library* performs returns only the data for which access has been granted by the user.

As shown in Figure 6, both the *AFP App* and *AFP Library* reside and operate at the application level of the Android stack. It is important to note that even when AFP is used, the Android OS is still fully enforcing its standard permission model. Hence, in its current implementation, AFP can be seen as an *enhancement* of the standard Android permission

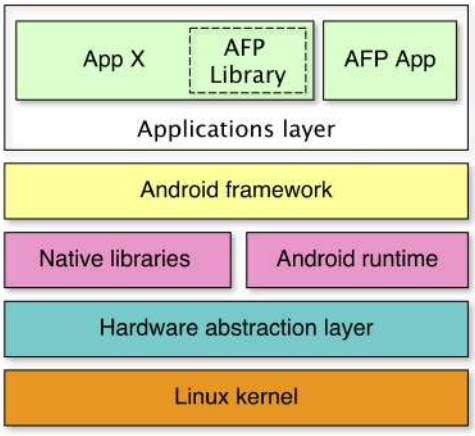


Fig. 6. Android software stack with added AFP components

system for privacy-aware users, rather than an alternative solution. This design choice of operating only at the application level of the Android stack, without modifying the lower levels, has been made in order to (i) allow for experimentation without the constraints introduced by the underlying levels, (ii) validate the effectiveness of feature- and level-based permissions prior to developing an implementation fully integrated with the Android platform and, (iii) simplify potential future porting of AFP on other platforms.

4 Implementation and Used Technologies

AFP makes use of a number of different techniques and technologies. Static analysis techniques are used to verify APK packages uploaded by developers on the AFP Server. Specifically, static analysis is utilized to verify that developers accessed sensitive resources only through the methods provided by the AFP Library (otherwise we cannot guarantee that the preferences set by users in permissions configurations will be fulfilled). In order to do so, AFP utilizes an intra-procedure analysis to detect Android API invocations within each method of the app's code. Our prototype implementation adopts the list of sensitive Android APIs provided by Rasthofer *et al.*, who also provide a methodology to obtain an up-to-date list. For the implementation of this analysis, we rely on the static analysis framework Soot [22], coupled with Dexpler [23] to disassemble APK packages and transform Android's Dalvik bytecode into a format suitable for analysis.

The integrity of the AFP Library implementation is also checked by the AFP Server through a checksum-based integrity verification mechanism. This permits to avoid the possibility that malicious developers could tamper with the library, thus circumventing the need of obtaining authorization against the permissions configuration.

Communication between the AFP App and any AFP-compliant app is enabled by Android's *Intents* [24]. An *Intent* is a special kind of object used to enable inter-app communication. In AFP, *explicit Intents* are leveraged to both redirect the end user to the AFP App and return the configured fine-grained permissions configuration once it has been personalized.

Changes to the permission system introduce the risk of app instability, as apps may not expect to have their permission requests denied [25]. When denying permissions leads to crashes, users are likely to become more permissive to improve app stability, thus counteracting the whole reasoning behind feature- and level-based permissions. With this concern in mind, in our implementation, we make use of

"mocking"[3]: in the event of a denied permission our system supplies apps with well-formed but non-sensitive data. 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. This enables apps to continue functioning usefully unless access to the protected resource is critical for its correct behavior.

To perform the automatic extraction of the Android components composing the mobile app (step 2 in Figure 1) we firstly decode the input APK using apktool[4]. The Android manifest file is then analyzed via a simple XML parser we developed in Java. The analyzer of the Java bytecode is implemented by using the Apache Commons Byte Code Engineering Library[5].

The tool that allows developers to specify feature-components mappings (step 3) has been implemented as a web-based tool, built upon the Flask[6] web framework.

The AFP Instrumenter is implemented by using several tools. The tool apktool is used for decomposing the APK and producing a Classes.dex file containing the app bytecode. Then, the dex2jar[7] tool is used to obtain a conventional jar file that, subsequently unpacked via the zip shell command, permits to obtain the .class files constituting the app. Rewriting of the .class files is done by our Java implementation of Algorithm 1, leveraging the Apache Commons Byte Code Engineering Library. Instrumented .class files are then repackaged back to a .dex archive via the Android SDK dx[8] tool, and the APK archive is reassembled using again apktool. At the end, the resulting package is signed using jarsigner[9]. The whole end-to-end process is tied together by a Python script.

5 Evaluation

In this section, we report the four independent experiments we performed to evaluate the AFP approach. For the purposes of the experiment 2, 3 and 4, we focused on the three Android APIs that are among the ones considered the most sensible by end users [27] while at the same time widely used by apps on the Google Play market [28]: Camera, LocationManager, and MediaRecorder. To allow easy replication and verification of the experiments, we provide a complete **replication package**[10] including: the source code of all the components of the AFP approach, the source code of the measuring tools we implemented for carrying on the experiments, the raw data we obtained from the experiments, and all the scripts for analysing the experiments' results.

3. This aspect is inspired by the Mockdroid approach by Beresford et al. [26])
4. http://ibotpeaches.github.io/Apktool/
5. http://commons.apache.org/proper/commons-bcel/
6. http://flask.pocoo.org
7. http://github.com/pxb1988/dex2jar
8. http://wing-linux.sourceforge.net/guide/developing/tools
9. http://docs.oracle.com/javase/tutorial/deployment/jar/signing.html
10. https://github.com/gianlucascoccia/androidflexiblepermissions

5.1 Experiment 1: Performance of the AFP instrumenter

Design – The *goal* of this experiment is to assess the performance of the AFP Instrumenter, the module of the AFP server that performs the app static analysis and instrumentation. We chose the AFP Instrumenter as the *object* of our experiment since (i) it is the most complex component in our AFP Server, and (ii) its malfunctioning or low performance in terms of execution time may negatively impact the adoption of the whole approach by developers, who will not be willing to spend (relatively) long time for the result of the app instrumentation phase. This experiment is designed as a multi-test within object study [29], because it is conducted on a single *object* (i.e., the current implementation of the AFP Instrumenter) across a set of *subjects* (i.e., the APKs archives). More specifically, we randomly selected 1,277 APK archives from a dataset consisting of 11,917 free apps from the Google Play Store; the dataset was 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 [30]. We executed the experiment by (i) automatically generating a feature-component mapping containing a feature for each Java class of the app (this can be considered a worst case scenario for our instrumenter), (ii) isolating the AFP Instrumenter component so that it could be programmatically executed in isolation, and (iii) sequentially executing AFP Instrumenter for all the 1,277 APKs. For each execution of the AFP Instrumenter, we measured the time for performing each single step of its internal pipeline (see Section 4). Measurements were taken via a Macbook Pro-Retina running Mac OSX 10.11.5 with a 2.6 GHz Intel core i5 processor and 8 Gb of memory.

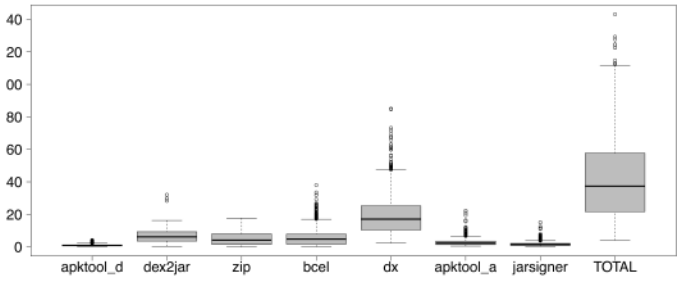


Fig. 7. Execution times of the AFP Instrumenter pipeline (in seconds)

Results – Figure 7 shows the execution times of each step of the AFP Instrumenter pipeline. Each step takes an average of less than 10 seconds, with the only exception of the dx tool (18.78 seconds in average), mainly because of its heavy I/O operations and performed optimizations[11]. When considering the total execution time of the whole pipeline (see the last box plot in the figure), we can observe that our AFP Instrumenter takes an average of 41.19 seconds to complete, with a minimum of 4.03 seconds and a maximum of 143.07 seconds.

11. https://android.googlesource.com/platform/dalvik/+/a9ac3a9d1f8de71bcdc39d1f4827c04a952a0c29/dx/src/com/android/dx/command/dexer/Main.java

Discussion – We consider the results as satisfactory. On average, developers have to wait less than a minute for obtaining the AFP-enabled app from the AFP Server, and less than 3 minutes in the worst case of our experiment. Since the AFP Instrumenter is executed only once for each app, the waiting time for developers due to app instrumentation is reasonable. Hence:

> The performance of the AFP Instrumenter is satisfactory, requiring on average less than a minute.

Threats to validity – A possible threat to validity of our experiment is represented by the selection of only free apps as subjects. Although this choice was driven by budgetary constraints, free apps are representative as they represent 75% of all Google Play Store apps and are downloaded more often [31] than paid apps.

A second threat is represented by the fact that, in this experiment we consider only one kind of *hardware* machine. This choice is mainly guided by budget constraints related to both time and available resources. Notwithstanding, the used hardware is consumer-grade, hence we believe that collected measures are representative of performances that can be obtained on ordinary hardware. Moreover, it is important to note that the AFP Instrumenter is deployed in the AFP Server, whose hardware and software performance can be far higher than the machine we used for this experiment and can be easily scaled up if deployed in a virtualized/containerized environment.

5.2 Experiment 2: Performance of AFP-enabled apps

Design – The *goal* of this experiment is to assess the performance of the AFP-enabled apps at runtime (i.e., app X in Figure 1). The main rationale behind this experiment is to assess how the application of our AFP approach may actually impact the performance of instrumented apps, thus potentially impacting the overall user experience.

For this experiment, we selected 7 mobile apps from a publicly available dataset composed of 2,443 open-source Android apps that are freely distributed in the Google Play store and whose source code is hosted on GitHub [32]; these two conditions permitted us to have a dataset with apps designed and developed as real projects with real users, and to easily check that instrumented apps behave as the original ones (we did this by a combination of source code inspection and the addition of logging instructions in key parts of the app). Among the 2,443 apps, we randomly selected the 7 apps (see Table 1) among those requesting at least one of the permissions considered for our experiments (i.e., geolocation, camera and microphone access).

We executed the experiment by performing the following steps for each app: (i) we defined one or two (depending on the complexity of the app, see Table 1) common usage scenarios that start from the main activity and end with the complete stop of the app; (ii) we executed each usage scenario, while measuring the CPU load and memory consumption of the process of the app; (iii) we created a feature-component mappings using the AFP Web application; (iv) we instrumented the app via the AFP Instrumenter by using the mapping defined in the previous step; (v) we executed and measured again each usage scenario on the instrumented version of the app.

TABLE 1
Selected apps for study 2

ID	Name	Version	Type	Scenarios
a_1	WordPress	5.6.1	Blog manager	2
a_2	Ottawa Bus Follower	2.0.11	Bus-related utility	2
a_3	Streetlight Seattle Reporter	1.2.0	Citizen participation	2
a_4	Local Weather	1.0.0.7	Weather	1
a_5	Run Helper	1.3	Fitness tracker	1
a_6	Selfie HD	1.1	Photo camera	1
a_7	Flickr Uploader	2.3.2	Photo upload	1

From a tooling perspective, we used (i) MonkeyRecorder[12] for recording all the actions we manually performed for each app during a pilot manual execution of all the basic usage scenarios, (ii) a shell script using Android monkeyrunner[13] for replaying the previously recorded scenarios, and (iii) another shell script periodically executing the Android top or dumpsys tools via the Android Debug Bridge (ADB)[14] for collecting the CPU load and memory consumption of the app at runtime. All the shell scripts and tools were executed from the same laptop of Experiment 1, whereas the apps have been executed on an LG G3 855 (quad-core CPU at 2.5 GHz and 3 Gb of memory) running Android 6.0.

Results – Collected measurements are presented in Figure 8. For each app, we performed a comparison of both the CPU load and memory consumption for its two versions (i.e., original or instrumented) by using the Mann-Whitney test [29] with $\alpha = 0.05$, one-tailed. In all cases, we obtained a p-value much larger than α, thus allowing us to confirm that there is low difference in the medians of CPU and memory consumption, with and without app instrumentation.

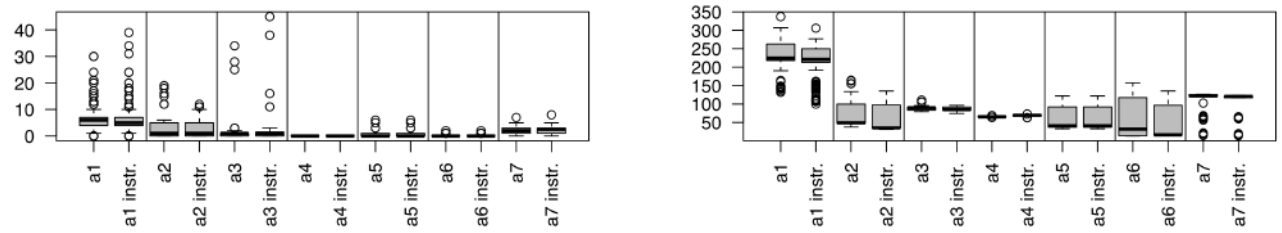
Discussion – From Figure 8 it is evident that both CPU load and memory consumption of the original and instrumented versions of each app are comparable, as confirmed by statistical tests. The results of this experiment give a positive indication about the performance of AFP-based apps.

It is important to note that the focus is not on the formal systematic assessment of the *precision* of the app instrumentation (i.e., we do not have a formal proof that instrumented apps do not crash in some corner cases); nevertheless, we performed a manual assessment of stability of the 7 apps in experiment 2 by performing a set of evaluation runs, observing that the instrumented app conformed to the expected behavior. To recap:

> Performance of AFP-enabled apps are comparable to those of regular apps.

Threats to validity – One common risk to validity of the experiment is the threat that adopted feature-component map-

12. https://android.googlesource.com/platform/sdk/+/6db5720/monkeyrunner/src/com/android/monkeyrunner/recorder/MonkeyRecorder.java
13. http://developer.android.com/studio/test/monkeyrunner
14. http://developer.android.com/studio/command-line/adb.html



(a) CPU load of selected apps (percentage)

(b) Memory consumption of selected apps (in Mb)

Fig. 8. Performance of selected apps: original (a) and instrumented (b)

pings and execution scenarios may not be representative of real app usage. To mitigate this threat, as a preliminary step, selected apps were examined by (i) analysing apps description in the Google Play store, (ii) manually inspecting their source code, and (iii) performing a set of preliminary runs while observing app behavior. Two different researchers were involved in the definition of both artifacts: first they were proposed by one and then verified to be representative by a second one.

A second threat is represented by the limited amount of apps involved in the experiment (7), a very small minority of all apps available to mobile app users. Hence, results might not generalize to other apps. Nonetheless, selected apps have reasonably varied purposes (see Table 1), thus partially mitigating the threat.

Finally, as for experiment 1, in this experiment we consider only one kind of *hardware* machine for the server and mobile device. As the hardware we used is likewise consumer-grade, we believe that collected measures are representative of those that can be obtained on ordinary hardware.

5.3 Experiment 3: Usability and acceptance of AFP by developers

Design – The *goal* of this experiment is to evaluate the usability and acceptance of AFP by developers. For this purpose, we conducted an on-line study, involving Android developers, in which we asked them to build the feature-component mappings for the apps they developed. We focused on this aspect as it is the main effort required to developers to make their app AFP-compliant (all the other steps are automated). This experiment is composed of three main phases:

1) **Recruiting:** we posted an announcement on pertinent on-line discussion groups (i.e., Android developer forums, mailing lists) to enlist developers willing to take part in the study. Each developer was asked to provide (at least) one link to an app published on the Google Play store, on which (s)he worked (either alone or in a team).
2) **Mapping:** the participants who provided a working link in the previous step were then invited to access a web-based app containing: (i) the definitions of feature- and level-based permissions, and (ii) the AFP web editor for the feature-component mappings. The participants were instructed to create the feature-component mappings for one of their previously-linked apps, and we collected the mappings defined by developers. We kept track of the time required by each participant for creating the feature-component mappings.
3) **Evaluation:** after completing the mapping, participants were asked to complete an on-line evaluation questionnaire about AFP.

In order to encourage developers in participating to the experiment, we informed them that, during the mapping step, they did not have to manually input the activities of their Android app, as we had preemptively loaded them in the web editor after downloading and analyzing the manifest file of each app collected in the recruiting step.

TABLE 2
Structure of the evaluation questionnaire used for developers

Evaluation goal	Question ID	Question text
Acceptance	q_1	*Is the definition of level-based and feature-based permissions clear to you?*
	q_2	*Do you rate the definition of level-based and feature-based Android permissions as useful?*
	q_3	*Are you willing to use feature- and level-based permissions in your apps?*
Usability	s_1-s_{10}	As defined in the System Usability Scale [33].

The structure of the evaluation questionnaire is shown in Table 2. The acceptance part is composed of three closed questions (q_1 - q_3) with possible answers ranging on a five-point scale. For the usability part, we adopted the System Usability Scale (SUS), a simple and widely-adopted scale for assessing the usability of products and services [33], [34]. The SUS consists in a questionnaire composed of 10 items (s_1 - s_{10}) and each item can be assessed by respondents along a 5-levels Likert scale ranging from *Strongly agree* (4 points) to *Strongly disagree* (0 points). SUS is proved to be a valuable, robust, and reliable evaluation tool and it correlates well with other subjectives measures of usability [33], [35].

Results – A total of 11 developers completed both the mapping definition and the evaluation questionnaire, providing

TABLE 3
Apps developed by participants of Experiment 3

Developer ID	App ID	Package Name	Version	Type	Relevant resources	# Features	Mapping time (s)
D_1	A_1	com.rubikssolutions.daily5sec	3.2	Video sharing	Camera, Microphone	2	21
D_2	A_2	com.ambiensvr.mobile	0.9	Augmented reality	Camera	4	924
D_3	A_3	fr.inria.es.electrosmart	1.6	Electromagnetic waves meter	Location	8	337
D_3	A_4	ums.lovely.university	9.8.4	University management system	Camera, Location	4	1809
D_4	A_5	com.digitech.foodel	1.9	Food delivery	Camera, Location	22	1758
D_5	A_6	com.mobile.wabi_tech.gadfly	2.2.9	Citizen participation	Location	3	49
D_6	A_7	com.Jitendra93266.jitu.knowmovies	1.0	Movies database	-	3	54
D_7	A_8	de.jw.mymensa	0.9.0	Menu viewer	-	2	47
D_8	A_9	com.peaklens.ar	1.0.11	Augmented reality	Camera, Location	7	455
D_9	A_{10}	com.yopapp.yop	1.9.4	Online marketplace	Camera, Location	4	192
D_{10}	A_{11}	com.myoxygen.press.weaf	3.0	Aerospace news	Location	4	61
D_{11}	A_{12}	ro.notnull.IdenticalFilesFinder	3.4.0	System management	-	4	80

us with twelve mappings in total (one developer performed the mapping construction for 2 apps). Developers participating in the experiment are also quite heterogeneous, both in terms of experience, number of developed apps, and size of organization. Specifically, participants have an average of 3.45 years of Android development experience (standard deviation = 2.66) and their majority (5) developed between 2 and 5 Android apps during their career, followed by 2 developers who developed more than ten. For what concerns organizations, the majority of developers work in small organizations (i.e., with 2 to 10 members), but we have also participants working in organizations with a number of members *between 2 and 10* and *between 10 and 50*. Finally, six developers declared to be *Satisfied* with current Android permissions, three declared to be *Unsatisfied* and two are *Unsure*.

A breakdown of the apps submitted by the participants is provided in Table 3. Minimum and maximum amount of features defined by developers is 2 and 22, respectively. For what concerns the time for creating the feature-component mappings, developers took an average of 482.25 seconds, i.e., 8.03 minutes (median = 136s, min = 21s, max = 1809s, SD = 660s). Even in the worst case, the time required by the participants to create the feature-activities mapping is close to half an hour. We consider such amount of time acceptable, considering that the definition of such mapping is conducted only once for an app.

Figure 9 summarizes the distribution of answers for questions q_1, q_2 and q_3. When developers were asked about whether they understood the concepts behind feature- and level-based permissions (q_1), only one developer answered *No*, three answered *Absolutely Yes* and the remainder *Yes*. On a similar note, when asked whether they consider feature- and level-based permissions as useful (q_2), answers were two *Absolutely Yes*, five *Yes*, one *Don't know*, two *No* and one *Absolutely No*. Concerning whether they would be willing to use the AFP permissions in their apps (q_3) answers were two *Absolutely Yes*, four *Yes*, two *Don't know*, two *No* and one *Absolutely No*. For all three questions median value of answers is *Yes*.

Results of the usability part of the evaluation questionnaire are presented in Figure 10, where each column of the heatmap represents the distribution of answers for each of the ten statements that comprise SUS. The procedure described in the SUS guidelines [33] was applied to normalize answers to each statement. For most statements, answers provided by respondents are mostly agglomerated towards the middle of the scale, with the exception of s_3 and s_4, skewed towards the upper and lower end of the scale respectively. A mean SUS score of 49.77 was obtained across all participants.

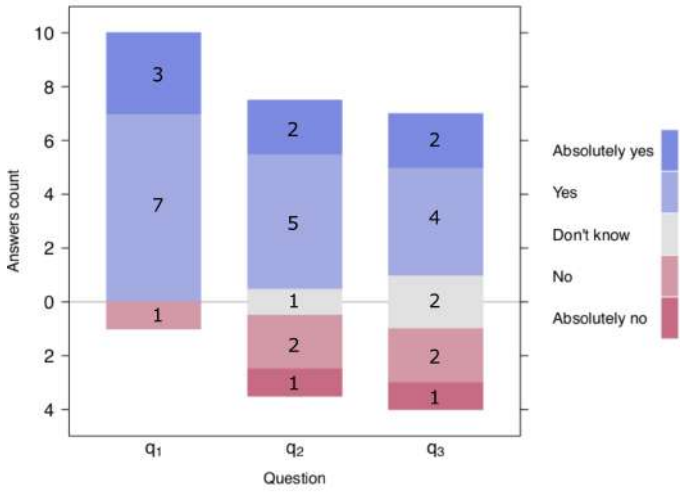


Fig. 9. Results about the acceptance of AFP by developers

Discussion – When considering the answers to the *acceptance* part of the evaluation questionnaire (presented in Figure 9), we can see that almost all developers understood the concepts required to be able to use feature- and level-based permissions (q_1). On a similar note, the majority of participants considered feature- and level-based permissions as useful (q_2). Summarizing:

> Feature- and level-based permissions are both understood and deemed useful by developers.

Concerning their willingness to adopt the proposed permissions approach in their apps (q_3 in Figure 9), the majority of developers declared that they would be willing to adopt it in their apps. Although the number of participants is

relatively limited, such results are encouraging. We can state that:

> The majority of the participants are willing to adopt feature- and level-based permissions in their apps.

Focusing on answers to the *usability* part of the evaluation questionnaire (shown in Figure 10), we can see that answers are clustered towards the middle of the scale for the majority of statements, revealing an acceptable usability. Further confirmation is found in the obtained mean SUS score (49.77) that, according to prior research [34], is to be evaluated as *Ok*. Exceptions are statements s_3 and s_4. Most participants are in agreement with the former, that reads *"I thought the system was easy to use"*, confirming that the process was not overly difficult. At the same time developers felt in agreement with the latter: *"I think that I would need the support of a technical person to be able to use this system"*. Such results, and the fact that developers are not always familiar with permission-related technical aspects [36], encourages us to investigate in the future on techniques to assist in (or even to automate) the definition of feature-component mappings. Summarizing:

> Developers positively evaluated the usability of the AFP approach. Improvements can be made on making the definition of the feature-component mappings more straightforward from a technical perspective.

Threats to validity – Possible threats to the validity and points of improvement for experiment 3 are as follows. The number of participants in the experiment (11) represents a very small minority of mobile app developers in the real world, hence results might not generalize. This threat is mitigated by the fact that developers who participated in the experiment have varied years of experience, nationalities and work in organizations of different size.

A second threat is represented by the limited number of apps for which developers created the mappings. Thus, the results of our study might potentially not generalize to other apps. This threat is mitigated by the fact that submitted apps have very different purposes, provide different features, and have been developed by different developers.

Finally, as the study has been conducted on-line, we had no way to ascertain that participants fully understood the task they were asked to complete. We mitigated this potential threat by directly asking in the final questionnaire whether developers had additional comments or doubts to clarify with respect to AFP.

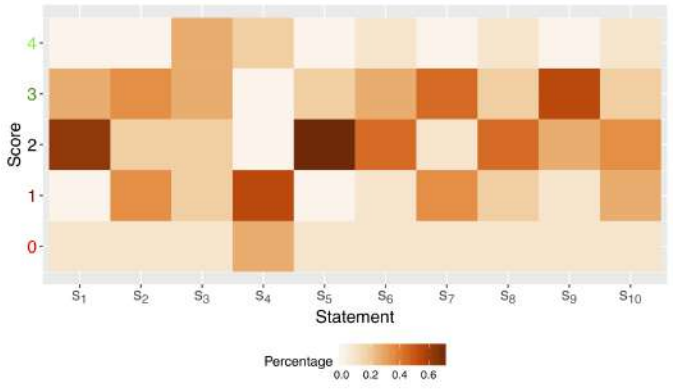


Fig. 10. Frequency distribution of answers to SUS statements by developers

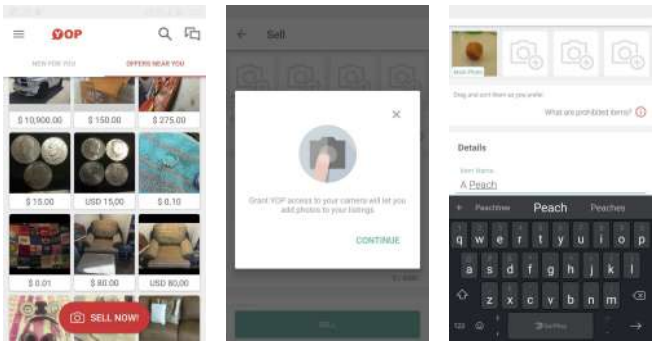
5.4 Experiment 4: Usability and acceptance of AFP by end users

Design – The *goal* of this study is to evaluate usability and acceptance of AFP from the end-user perspective. We conducted an in-person study involving 47 participants. This experiment is composed of three main phases:

1) **Pre-study:** each participant was given a short description about the goal of the experiment, together with the definition of feature- and level-based permissions. Demographic information was also collected.
2) **First trial:** each participant was asked to try out three Android apps, employing either AFP or the current Android permission system. An evaluation questionnaire was given at the end of the trial.
3) **Second trial:** the participant was asked to repeat the trial, this time employing the permission system that was not used during the previous phase. Again, an evaluation questionnaire was given at the end of the trial.

During each trial, participants were allowed to freely explore the given apps, while being monitored by one researcher that provided assistance, when needed. The researcher ensured that a minimal set of steps, namely an *execution scenario*, was executed for each app, to guarantee that participants were properly familiar with the apps and the underlying permissions systems before filling out the questionnaires. Each execution scenario was defined a priori and focusses on one of the app main functionalities. An example of execution scenario for app A_{10} is given in Figure 11: in order to sell an object on the marketplace, users have to (a) tap on the "sell now" button, (b) take a picture of the object with the device camera (granting the required permissions if needed) and (c) fill out remaining listing data before submitting. Users were asked to complete the same execution scenarios between the two trials although inevitably some middle steps differed, i.e., users had to grant permissions at runtime under Android permissions as opposed to app startup with AFP. All trials were performed on a device specifically made available, namely a LG G3 running Android 6.0.

In order to keep the experiment as representative as possible, we decided to reuse three of the apps for which real developers provided a mapping in Experiment 3. The app selection was performed with the goal of having at least one app making use of each of the device resources currently adopted by the current implementation of AFP (i.e., Camera, Microphone, and Location). Unfortunately, we could not successfully instrument app A_1, the only one in our dataset that uses the microphone, because it relies on Java reflection (a language construct traditionally hard to deal with by approaches relying on static analysis [37]).



(a) (b) (c)

Fig. 11. Example of execution scenario for the com.yopapp.yop app: to sell an object the user (a) taps on the "sell now" button, (b) takes a picture and (c) fills out listing details

Hence, we discarded it and selected an alternative app, thus ending with apps A_3, A_{10}, A_{11} in our final app selection.

A breakdown of the structure of the evaluation questionnaires provided to users is shown in Table 4. It can be divided into three main parts, according to the different goal of each one: the first part comparatively evaluates AFP and Android *trustability*, the second one evaluates the *acceptance* of AFP and the last one assesses *usability* of AFP.

TABLE 4
Structure of the evaluation questionnaire for end users

Evaluation goal	Evaluated for	Question ID	Question text
Trustability	AFP vs Android	Q_1	*How do you judge the way the app asked for permissions?*
		Q_2	*How likely are you to use the app on your device, considering the permissions you were asked for?*
Acceptance	AFP	Q_3	*Is the definition of feature-based permissions clear to you?*
		Q_4	*Is the definition of level-based permissions clear to you?*
		Q_5	*Do you rate feature-based permissions as useful?*
		Q_6	*Do you rate level-based permissions as useful?*
Usability	AFP	S_1-S_{10}	As defined in the System Usability Scale [33].

The first part of the questionnaire (*Trustability*) consists of two questions (Q_1, Q_2). It was filled by participants at the end of both the second and the third phase of the experiment. The second part of the questionnaire (*Acceptance*) contains four questions (Q_3 - Q_6) and it was filled by participants after completing the third phase of the experiment. All answers to questions Q_1 – Q_6 range on a five-point Likert scale.

Also the third part of the questionnaire (i.e., *Usability*) was filled by participants after completing the third phase of the experiment. In this part, we relied again on the System Usability Scale (S_1 - S_{10}). At the end of the questionnaire, open comments were collected from the participants.

Results – Professions and backgrounds of participants are varied and include students, shop assistants, mechanical engineers, lawyers, etc. Participants were mostly male (68%) and and the mode of their age is *Between 21 and 30 years old*. Although not perfectly balanced, we consider the group of participants to be a good sample of the general smartphone users population, itself skewed towards younger ages and males [38]. The mean self-assessed knowledge of Android is 3.43 on a 1-5 scale (1.17 standard deviation). Roughly half of the participants (24/47) tried out the apps with the Android permission system as first, and with AFP as second. The opposite order was adopted for the others.

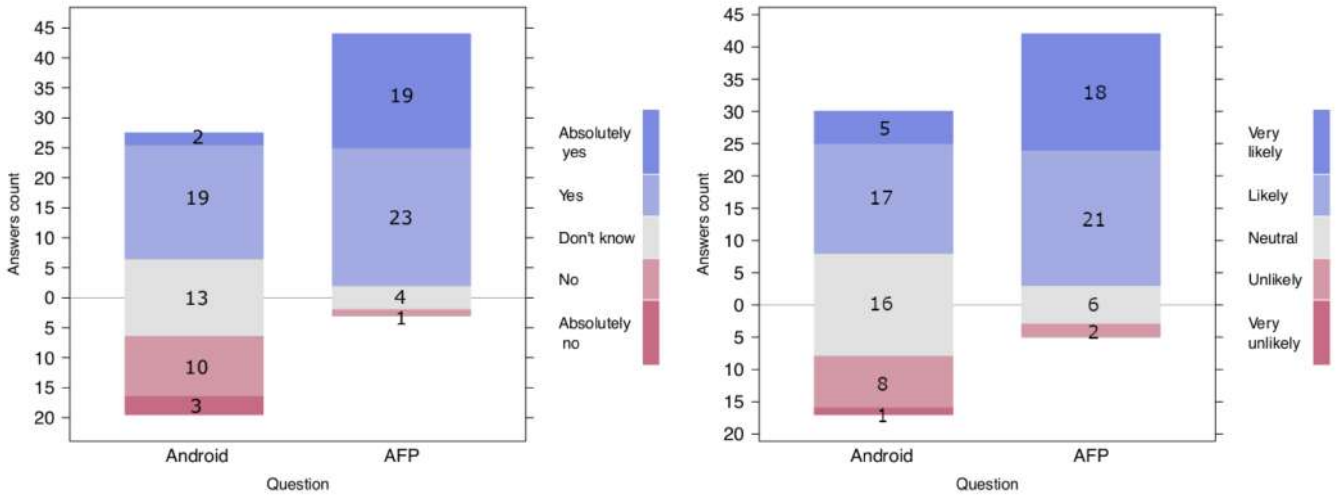
Table 5 provides the breakdown of the permission preferences configured by participants during the experiment. Overall, preferences were varied, with each access level being selected by at least one participant for all features. On average, participants required 17.27 seconds to configure their permission preferences for A_3, 36.45 seconds for A_{10} and 13.63 seconds for A_{11} (with a standard deviation of 13.21, 25.08 and 12.64 seconds respectively).

TABLE 5
Breakdown of end users privacy preferences

App	μ Configuration time (σ)	Feature	Resource	Access level (%)	
A_3	17.27s (13.21s)	F_1	Location	Full access:	19 (40%)
				Region only:	6 (13%)
				City only:	17 (36%)
				Deny:	5 (11%)
A_{10}	36.45s (25.8s)	F_2	Camera	Allow:	38 (81%)
				Deny:	9 (19%)
		F_3	Location	Full access:	4 (9%)
				Region only:	17 (36%)
				City only:	23 (49%)
				Deny:	3 (6%)
		F_4	Location	Full access:	5 (11%)
				Region only:	20 (43%)
				City only:	18 (38%)
				Deny:	4 (9%)
A_{11}	13.63s (12.64s)	F_5	Location	Full access:	9 (19%)
				Region only:	15 (32%)
				City only:	10 (21%)
				Deny:	13 (28%)

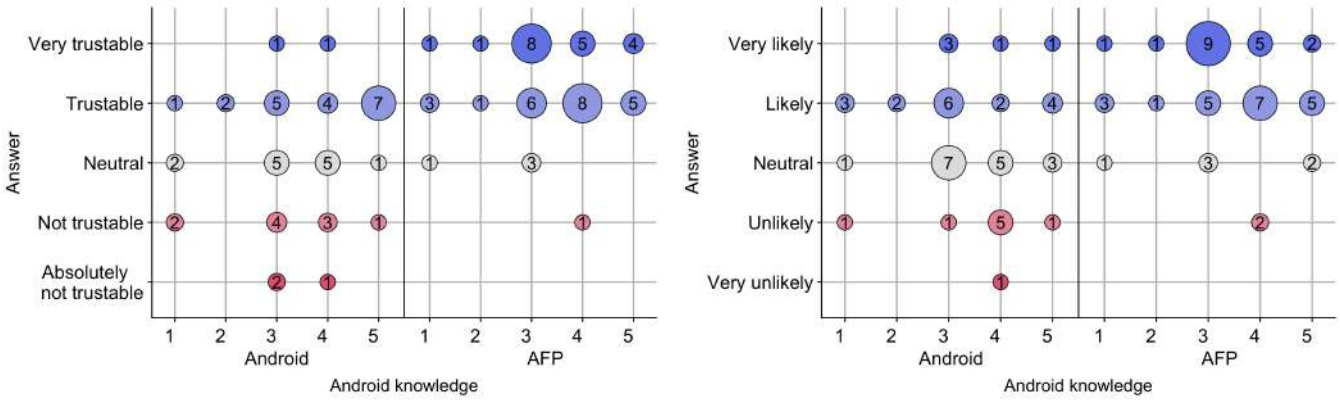
Figure 12 summarizes the distribution of answers for questions Q_1 and Q_2 for both AFP and the Android permission system (recall that participants were asked these questions twice). For both questions, users provided more favourable answers for AFP, with a median value of *Trustable* for Q_1 and *Likely* for Q_2, as opposed to Android which achieved a median of *Neutral* for both questions. Obviously, differences in answers are statistically significant, which we confirmed by performing the two-tailed Mann-Whitney U-test [39]. We obtained a p-value of $7.623e^{-08}$ for Q_1 and $4.802e^{-05}$ for Q_2, thus rejecting the null hypothesis that the distributions of the answers about Android and AFP are equal. Figure 13 provides a breakdown of answers to Q_1 and Q_2 by self-declared Android knowledge of participants. For both questions, AFP achieved more favourable answers even for participants with a lower level of knowledge.

Answers for the Acceptance part of the questionnaire (i.e., Q_3, Q_4, Q_5 and Q_6) are summarized in Figure 14. The answers are skewed towards the positive part of the



(a) Perceived trustability (Q_1) (b) Likeliness to use the app (Q_2)

Fig. 12. Perceived trustability of Android and AFP permission systems w.r.t. the way the app asked permissions (Q_1) and how likely the participant is to use the app (Q_2)



(a) Perceived trustability (Q_1) (b) Likeliness to use the app (Q_2)

Fig. 13. Perceived trustability (Q_1) and likeliness to use the app w.r.t. the way the app asked permissions (Q_2) by self-declared Android knowledge

scale and the median value is *Absolutely yes* for most of them, with the only exception of Q_5 for which the median value is *Yes*. Figure 15 provides an overview of answers to the same questions by self declared Android knowledge of participants. For all levels of knowledge, the majority of answers fall in the upper end of the scale.

Results of the usability evaluation are shown in Figure 17. Each column of the heatmap presents the frequency distribution of answers for one of the ten SUS statements. The procedure described in the SUS guidelines [33] was applied to normalize answers to each statement. For all ten statements, participants provided mostly positive answers with the total amount of negative ones being less than 15% for all statements. A mean SUS score of 78.19 was obtained across all participants. Figure 16 goes deep in the perceived usability of AFP with respect to self-declared Android knowledge (note that, for odd numbered statements, higher is better; the opposite is true for even numbered ones). For all statements, no major differences can be observed in the distribution of the answers along knowledge levels.

Discussion – From the collected users privacy preferences (see Table 5), we can notice that choices are varied and, for all features, each level was selected by at least a minority of users. This variability in choices reveals that participants indeed took advantage of the added control provided by feature- and level-based permissions. Such data substantiates the intuition that users have diverse privacy concerns that cannot really be satisfied by one-size-fits-all approaches, like the current Android permission model. Hence, we can infer that:

> Feature- and level-based permissions allow for an experience more tailored to individual privacy preferences.

Still concerning users choices, it is worth noting that, when possible, users favoured more restricted access levels and seldom provided full location access. An exception is represented by feature F_1, for which full access is the

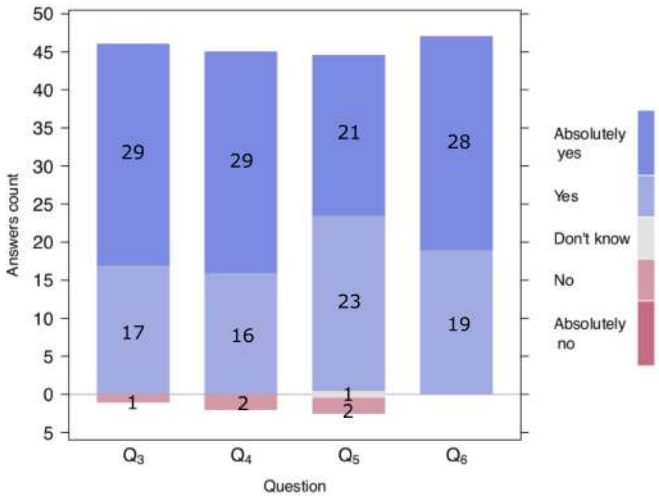


Fig. 14. Acceptance of AFP by end users in terms of: clarity of the definitions (Q_3 and Q_4) and usefulness (Q_5 and Q_6) of feature-based and level-based permissions).

consensus choice. During the execution of the experiment, we observed that users believed that full access was strictly necessary for performing this feature, thus explaining such difference. Consequently, we can state that:

> When possible, most users choose levels that disclose a restricted amount of data.

From the answers to questions Q_1 and Q_2 (see Figure 12), we can point out that users generally felt more secure when employing AFP and praised the added control over shared personal data provided by it. Figure 13 provides an indication that this preference towards AFP is constant across varying levels of Android knowledge. Such considerations are also supported by some of the comments collected during the evaluation questionnaire: one participant stated that *"(Between the two systems) I prefer AFP as I have more control on permissions and I feel more secure"*, while another participant pointed out that *"as a user, with AFP I am more aware on how an app uses my phone"*. This leads us to the following reflection point:

> Users feel more secure and are more willing to use apps on their smartphone when using feature- and level-based permissions.

Figure 14 outlines the distribution of answers to the acceptance part of the evaluation questionnaire (questions Q_3 through Q_6). The definitions of feature- and level-based permissions were both well understood and deemed useful by users, although the former achieved worse answers regarding its perceived usefulness (Q_5). We conjecture this difference is due to the fact that the perceived benefit of level-based permissions is more immediate to users. Figure 15 suggests that, also in the case of acceptance, the distribution of the answers is unaffected by the users' knowledge of Android. Further confirmation of the perceived usefulness can be found in the open comments: one participant stated *"I appreciated the greater choice of options provided by AFP"*; a second one noted that *"(Android) permission pop-ups are misleading and enforce a binary choice. I appreciated AFP's level-based permissions"*. In conclusion:

> Feature- and level-based permissions are both well understood and deemed useful by end users, although the usefulness of the latter is more immediate to them.

Regarding the results of the usability evaluation, according to prior research [34], the achieved mean SUS score (78.19) is to be interpreted as a *Good* level of usability (see Figure 17). Focusing on the distribution of answers for each statement, positive answers are the majority for all statements, even for users with a limited knowledge of the Android platform (see Figure 16). Nonetheless some points of improvement can be identified. Statements S_1, S_2, S_3 and S_5 achieved a comparatively lower amount of maximum score answers, revealing that some users did experience some difficulties while using the system. Investigating the comments left by participants, we noticed that some users would prefer to grant feature- and level-based permissions at runtime. In particular, one user noted *"I would like to grant permissions when needed. Configuring permissions preemptively could take too long for some apps"*. To address this issue, in the future we plan to investigate alternative ways to elicit user's privacy preferences (see Section 8). On the positive side, for S_{10}, mostly maximum score answers were collected, hence highlighting that participants did not consider the amount of new notions that they had to learn as excessive. Summarizing:

> AFP usability from the end-users perspective is evaluated as *Good*.

Threats to validity – There are several threats and points of improvement for Experiment 4. Although users were instructed to act as they would do with their own smartphone, they were still performing the trials in a controlled environment, potentially different from the normal. This means that participant activities and answers may differ from what can be observed in the real world. As future work, we will mitigate this potential threat to validity by performing the experiment via an app that users can install on their own smartphone in order to monitors the AFP-enabled apps during their usage.

Despite our efforts to have a balanced and unbiased set of participants, we ended up with a group of relatively young people (age *between 21 and 30 years old*) and with a majority of male participants (32 males as opposed to 15 females). Moreover, we are aware that the sample size of this experiment (47 participants) is limited with respect to all Android smartphone users today. We mitigated this potential threat by contacting participants with different backgrounds and professions, different experience about the Android platform, and by letting them interact with the real apps instead of Android emulators or simulated environments.

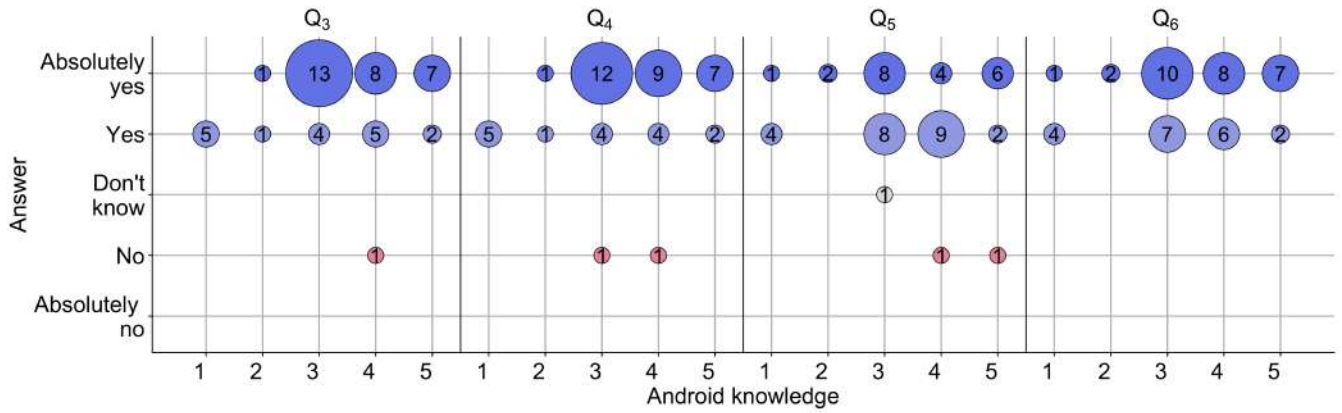


Fig. 15. Acceptance of AFP Q_3-Q_6 by self-declared Android knowledge

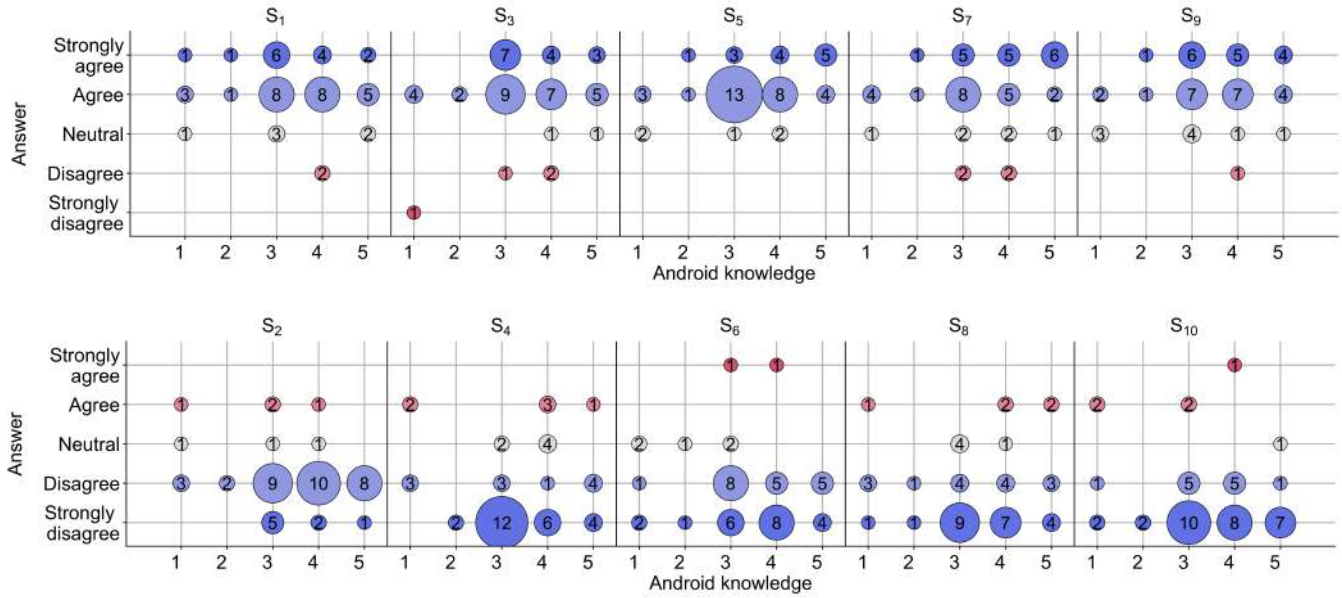


Fig. 16. Usability of AFP (S_1-S_{10}) by self-declared Android knowledge. For odd numbered statements higher is better. For even numbered statements lower is better.

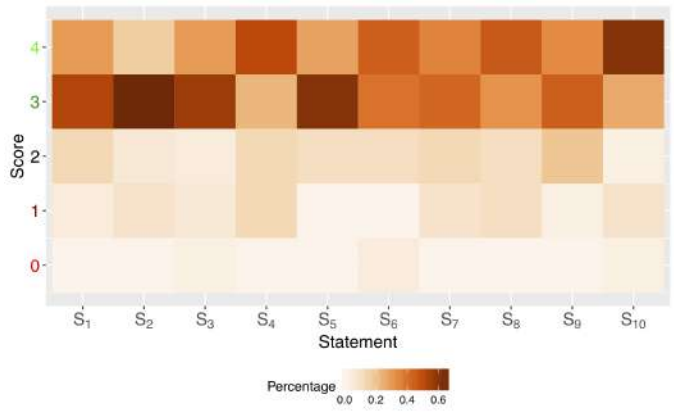


Fig. 17. Frequency distribution of answers to SUS statements by users

Another possible threat is represented by the limited amount of apps used to perform trials in the study. Hence, results of our study might potentially not generalize to other apps. We mitigated this threat by selecting three apps with very different purposes and features, thus collecting data on users behavior in varied scenarios. Additionally, participants were not familiar with selected apps in advance. Hence, their behavior during the trial might properly resemble the one of a user that runs a freshly installed app for the first time.

6 Limitations and Future Work

The proposed approach suffers a number of limitations that we plan to address in the future. At the time of writing, AFP only permits the specification of *PolicyItems* that reference a single resource. Our plan is to enhance the permission model by introducing consistency-checking of policies and

reasoning on multiple resources (e.g., do not allow access to contacts book if the app also requests Internet access).

As previously discussed in Section 3.6, our current implementation of AFP is fully contained in the application layer of the Android stack. Our prototype implementation was developed with the main goal of assessing the perception of users on the proposed feature- and level-based permissions. We are aware that a full market-scale adoption requires a more tightly-coupled integration with the lower levels of the Android stack. In this direction, the integration of AFP with existing works on kernel-space security [40], [41], [42], [43], [44] would be beneficial.

Furthermore, API wrappers are currently developed on a per-API basis. In the future, we plan on investigating possible categorizations of Android APIs, in order to group together similar ones, hence reducing the effort required to develop new wrappers. As these groupings must be well understood by both end users and developers, we plan on conducting new experiments to assess their effectiveness.

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. To address this issue, we are currently investigating a hybrid approach combining our static analysis with dynamic flow analysis. We have preliminary evidence that such a hybrid approach may result in a valid and viable compromise towards mitigating this challenge, as well as known issues of bytecode rewriting [45].

Another future work concerns assisting end users in the configuration of flexible-permissions. Since runtime permissions expose context, which can help users in making their decision, the extension would require to understand how the context can be suitably extracted and presented to users during the apps configuration phase, or even leveraged to automatically configure permissions without user intervention.

Another possible enhancement concerns the enactment of a greater integration with the Google Play Store, in order to favor adoption by end-users and developers. This can be achieved by using Google own APIs that allow third-party services to directly publish mobile apps to the Google Play Store.

Security of the AFP Server can also be enhanced implementing an authorization mechanism. OAuth2[15], an open communication protocol through which third-party service can securely manage authorized access to sensitive data, can be used for this purpose.

We are currently defining a procedure for (semi-) automatically extracting the features provided by an Android app from its binary or source code.

7 Related work

Android permissions and their usage have attracted considerable research interest over the years. In [3], Felt et al. report on the usability study they performed to evaluate whether Android users pay attention and understand permissions upon installing apps. The outcome of the study is that Android users make security and privacy decisions with a minimal comprehension of what the implications of their decisions will be.

The problem addressed in our paper falls into the class of problems identified in [3], and from there it takes the move. The authors of [7] investigated whether users are willing to pay premiums for privacy: results suggest that many smartphone users are concerned with their privacy and are willing to pay premiums for apps that are less likely to request access to personal information. Similar conclusions are presented by Kelly et al. in an independent study [15] that, in the form of a series of semi-structured interviews, was aimed at understanding user's attention to and comprehension of permissions screens displayed during app's installation. Their findings highlight that, while permissions displays are generally viewed and read, mostly often they are not understood by Android users.

Recent work has focused on studying the new Android runtime permission system. In [8], we identified points of possible improvement in the new system analyzing mobile apps user reviews. In [46], Peruma et al. conducted an in-person study to investigate user perception of the previous, current and a new proposed system. Among their findings there is the indication that current system does not make users feel more secure than the old one.

Andriotis et al. studied users' adaptation to the new runtime permission model [47], collecting and analyzing anonymous data from 50 participants. Although their analysis indicates that users adapted positively to the new model, participants of their study hardly ever utilized the newer capabilities that allow a more efficient control over resources used by applications [17]. In [48], the same authors suggest that users should be informed about the resources needed by an app to provide its core functionalities before installation.

Many approaches aim to offer a finer-grade control over permissions using a wide array of strategies [26], [49], [50], [51], [52]. In particular, Mockdroid [26], TISSA [51] and SHAMDROID [49] allow users to restrict access to a resource by providing mocked information in place of real one whenever access to the resource is attempted. The evaluation presented shows that the majority of apps are able to continue execution when injected with mocked information but their usefulness might be greatly reduced for the end user. In [50], Jeon et al. define some strategies for developing fine-grained variants of Android permissions and implement Dr. Android, a prototype tool that allows to retrofit existing Android apps to make use of fine-grained permissions. Proposed by Nauman et al., Apex [52] allows a user to selectively grant permissions to apps based on runtime contextual information, such as the location of the device or the number of times a resource has been previously used. To our knowledge, AFP is the first approach that allows users to specify permissions on a per-feature basis.

SmarPer [53] couples contextual information with machine learning techniques for automatic permission-granting, mimicking users' decision while removing the disruption of permission request pop-ups. Analogously, Wijesekera et al. investigated the effectiveness of such techniques on a larger scale [54], [55], to assess current performance and practical limitations in actual implementations. Even though not directly comparable to our approach, we plan to

15. https://oauth.net/2/

investigate how to integrate similar automatic permission granting mechanisms in AFP in order to further assist end users during permissions configuration.

Many static analysis approaches for Android apps exist in the literature [56], [57], [58], [59]. Most of the approaches have been aimed at the detection of malware apps, analysis of how control [56] or sensitive data [57], [58] flows through Android apps to identify vulnerable or suspicious behavior, etc. AFP instead aims at addressing issues perceived by end users in the current permission systems. Shen et al. in [59] propose an extension of Android permissions named "Flow Permissions", directly inferred from an existing app using static analysis techniques, whose aim is to inform users on the existence of data-flows that could potentially leak sensitive information to the outside. Unlike our approach, permissions are statically analyzed for informative purposes only and not for the purpose of actually be enforced at runtime.

Aurasium [40] is an application-hardening service that permits to attach user-level sandboxing and policy enforcement code to arbitrary Android applications. Similarly to our approach, Aurasium employs repackaging to bypass the need of modifying the Android operating system. However, differently from AFP, permissions are still granted to the whole application monolithically.

More closely related to our approach is AWare [60], by Petracca et al. Similarly to AFP, AWare limits the scope of granted permissions. Unlike our approach, where scope is limited by features boundaries, AWare binds permissions to specific user inputs. Evaluated experimentally, AWare was found to be effective as an additional layer of defense against untrusted applications with potentially malicious purposes, while limiting the explicit authorization overhead. COMPAC [41] extends Android permission model in order to allow developers and users to assign only a subset of an application's permissions to some of the application's components. Unlike AFP, COMPAC requires modifications to the Android framework and kernel.

The work in [61] proposes a static taint analysis method to identify in Android sources what are the sensitive APIs through which sensitive data flows. Although not strictly related to our approach, the idea of identifying data flows that originate from sensitive sources together with the extraction of data related to app features [62], [63], [64] would permit our approach to (semi) automatically map features to the Android components that implement them.

In [43], the authors systematically studied 1,139 Android kernels and all the recent critical vulnerabilities. The authors propose KARMA, an adaptive live patching system for Android kernels. To protect kernel vulnerabilities from exploits, KARMA leverages a multi-level adaptive patching model that permits to filter malicious inputs by placing patches at multiple levels in the kernel. Thousands of Android devices are seamlessly supported. Authors of InstaGuard [44] instead propose a new approach that allows for instant deployment of hot-patches for mobile devices. The kernel vulnerabilities identified in [43], [44], together with the related kernel-level considerations therein, would be of undoubted interest towards a native low-level integration of the AFP approach directly in the Android operating system.

Researchers have investigated the effectiveness of *nudging*, i.e., the act of assisting individuals' privacy and security choices with soft paternalistic interventions that drive users toward more beneficial options [65]. Integrating such techniques into AFP can potentially be beneficial for end users. In [66] Almuhimedi *et al.* report on a study that evaluates the benefits of giving users an app permission manager and sending them nudges intended to raise their awareness of the data collected. Nudges proved to be beneficial, with 95% of participants reassessing their permissions, and 58% of them restricting some. In our approach, the role of permissions manager is performed by the AFP *App*. In [67], Balebako *et al.* performed a series of experiments to examine how timing impacts the salience of smartphone app privacy notices. Notices displayed during app use showed significantly increased users' recall rates over notices displayed in the app store. Liu [68] *et al.* implemented and evaluated a Personalized Privacy Assistant (PPA) to automatically recommend permissions to users based on prior answers to a small set of questions. In their experiments, 78.7% of the recommendations were adopted by users and the PPA was perceived as useful and usable. Introducing a similar assistant in the AFP *App* can ease the permission configuration process for end users.

8 Conclusions

In this paper, we have presented an approach aimed at overcoming some limitations of the current Android permission model. At the core of the proposed approach lies a new flexible permission model for Android apps in which end users can grant and negotiate the level of each single permission of a mobile app, on a per feature-basis.

The new permission model is supported by an infrastructure comprising three main components: a library internal to the apps enacting the approach at runtime; a stand-alone mobile app that allows end users to configure and negotiate at any time the permissions for each app on their devices; a web-based server for allowing developers to analyze their own mobile apps, and enhance them with the new flexible permission system with very low effort.

Evaluation of proposed approach has been conducted by designing, conducting, and reporting on four independent experiments aimed at empirically investigating key aspects of it: performance of the instrumenter, performance at runtime of instrumented apps, usability and acceptance of the approach for both end users and developers. Results are promising and provide confidence in the approach.

References

[1] Adam Lella, Andrew Lipsman, Ben Martin, "The Global Mobile Report: How Multi-Platform Audiences and Engagement Compare in the US, Canada, UK and Beyond," 2015, comsCore white paper.

[2] University of Alabama at Birmingham Online Masters in Management Information Systems, "The Future of Mobile Application," 2014, http://businessdegrees.uab.edu/resources/infographic/the-future-of-mobile-application/.

[3] 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*, 2012, pp. 3:1–3:14.

[4] S. Chitkara, N. Gothoskar, S. Harish, J. I. Hong, and Y. Agarwal, "Does this app really need my location?: Context-aware privacy management for smartphones," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, p. 42, 2017.

[5] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," 2018.

[6] M. E. Kummer and P. Schulte, "When private information settles the bill: Money and privacy in google's market for smartphone applications," 2016.

[7] 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*, 2013, pp. 211–236.

[8] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi, "An investigation into android run-time permissions from the end users' perspective," 2018.

[9] Z. Fang, W. Han, and Y. Li, "Permission based android security: Issues and countermeasures," *computers & security*, vol. 43, pp. 205–218, 2014.

[10] Y. Zhauniarovich and O. Gadyatskaya, "Small changes, big changes: an updated view on the android permission system," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 346–367.

[11] M. Madden, L. Rainie, K. Zickuhr, M. Duggan, and A. Smith, "Public perceptions of privacy and security in the post-snowden era," *Pew Research Center*, vol. 12, 2014.

[12] "ICT Leit Work Programme 2018-2020, 13 November 2018," 2018. [Online]. Available: http://ec.europa.eu/research/participants/data/ref/h2020/wp/2018-2020/main/h2020-wp1820-leit-ict_en.pdf

[13] P. Bourque, R. E. Fairley *et al.*, *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[14] G. L. Scoccia, I. Malavolta, M. Autili, A. Di Salle, and P. Inverardi, "User-centric android flexible permissions," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 365–367.

[15] P. G. Kelley, L. F. Cranor, and N. Sadeh, "Privacy as part of the app decision-making process," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013, pp. 3393–3402.

[16] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, "A conundrum of permissions: Installing applications on an android smartphone," in *Proceedings of the 16th International Conference on Financial Cryptography and Data Security*, 2012.

[17] P. Andriotis, G. Stringhini, and M. A. Sasse, "Studying users' adaptation to android's run-time fine-grained access control system," *Journal of information security and applications*, vol. 40, pp. 31–43, 2018.

[18] S. Motiee, K. Hawkey, and K. Beznosov, "Do windows users follow the principle of least privilege?: Investigating user account control practices," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, 2010, pp. 1:1–1:13.

[19] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness." in *USENIX security symposium*, vol. 13, 2013.

[20] R. Böhme and J. Grossklags, "The security cost of cheap user interaction," in *Proceedings of the 2011 New Security Paradigms Workshop*. ACM, 2011, pp. 67–82.

[21] N. S. J. Lin, B. Liu and J. I. Hong, "Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings," in *Symposium On Usable Privacy and Security (SOUPS)*, 2014, pp. 199–212.

[22] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999, pp. 13–.

[23] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, 2012, pp. 27–38.

[24] "Android developers," http://developer.android.com/training/permissions/requesting.html.

[25] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. S. Wang, Z. Qian, and H. Chen, "revdroid: Code analysis of the side effects after dynamic permission revocation of android apps," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 747–758.

[26] 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*, 2011, pp. 49–54.

[27] A. P. Felt, S. Egelman, and D. Wagner, "I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12, 2012, pp. 33–44.

[28] P. H. Chia, Y. Yamamoto, and N. Asokan, "Is this app safe?: A large scale study on application permissions and risk signals," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12, 2012, pp. 311–320.

[29] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, ser. Computer Science. Springer, 2012.

[30] 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)*. Institute of Electrical and Electronics Engineers (IEEE), 2015, pp. 56–59.

[31] I. Gartner, "Gartner says free apps will account for nearly 90 percent of total mobile app store downloads in 2012," 2012, http://www.gartner.com/newsroom/id/2153215.

[32] T. Das, M. Di Penta, and I. Malavolta, "A quantitative and qualitative investigation of performance-related commits in android apps," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 443–447.

[33] J. Brooke *et al.*, "Sus-a quick and dirty usability scale," vol. 189, no. 194. London–, 1996, pp. 4–7.

[34] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability scale," vol. 24, no. 6. Taylor & Francis, 2008, pp. 574–594.

[35] J. Kirakowski and M. Corbett, "Sumi: The software usability measurement inventory," *British journal of educational technology*, vol. 24, no. 3, pp. 210–212, 1993.

[36] R. Balebako, A. Marsh, J. Lin, J. I. Hong, and L. F. Cranor, "The privacy and security behaviors of smartphone app developers," 2014.

[37] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 241–250.

[38] A. Smith, "Smartphone ownership–2013 update," *Pew Research Center: Washington DC*, vol. 12, p. 2013, 2013.

[39] G. W. Corder and D. I. Foreman, *Nonparametric statistics: A step-by-step approach*. John Wiley & Sons, 2014.

[40] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 539–552.

[41] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in android," in *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 2014, pp. 25–36.

[42] A. Dawoud and S. Bugiel, "Droidcap: Os support for capability-based permissions in android," 2019.

[43] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *26th Security Symposium USENIX*, 2017, pp. 1253–1270.

[44] Y. Chen, Y. Li, L. Lu, Y. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018.

[45] H. Hao, V. Singh, and W. Du, "On the effectiveness of api-level access control using bytecode rewriting in android," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 25–36.

[46] A. Peruma, J. Palmerino, and D. E. Krutz, "Investigating user perception and comprehension of android permission models," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 2018, pp. 56–66.

[47] P. Andriotis, M. A. Sasse, and G. Stringhini, "Permissions snapshots: Assessing users' adaptation to the android runtime permis-

sion model," in *Information Forensics and Security (WIFS), 2016 IEEE International Workshop on*. IEEE, 2016, pp. 1–6.

[48] P. Andriotis, S. Li, T. Spyridopoulos, and G. Stringhini, "A comparative study of android users' privacy preferences under the runtime permission model," in *International Conference on Human Aspects of Information Security, Privacy, and Trust*. Springer, 2017, pp. 604–622.

[49] L. Brutschy, P. Ferrara, O. Tripp, and M. Pistoia, "Shamdroid: Gracefully degrading functionality in the presence of limited resource access," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 316–331.

[50] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. android and mr. hide: Fine-grained permissions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012, pp. 3–14.

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

[52] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010, pp. 328–332.

[53] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J.-P. Hubaux, "Smarper: Context-aware and automatic runtime-permissions for mobile devices," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 1058–1076.

[54] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 1077–1093.

[55] P. Wijesekera, J. Reardon, I. Reyes, L. Tsai, J.-W. Chen, N. Good, D. Wagner, K. Beznosov, and S. Egelman, "Contextualizing privacy decisions for better prediction (and protection)," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, p. 268.

[56] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, 2014.

[57] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," 2009.

[58] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 106–117.

[59] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information flows as a permission mechanism," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 515–526.

[60] G. Petracca, A.-A. Reineh, Y. Sun, J. Grossklags, and T. Jaeger, "Aware: Preventing abuse of privacy-sensitive sensors via operation bindings," in *26th USENIX Security Symposium*, 2017, pp. 379–396.

[61] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 426–436.

[62] M. Harman, Y. Jia, and Y. Zhang, "App store mining and analysis: MSR for app stores," in *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, 2012, pp. 108–111.

[63] Y. Guo, Y. Li, Z. Yang, and X. Chen, "What's inside my app?: understanding feature redundancy in mobile apps," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 266–276.

[64] H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong, "Understanding the purpose of permission use in mobile apps," *ACM Transactions on Information Systems (TOIS)*, vol. 35, no. 4, p. 43, 2017.

[65] A. Acquisti, I. Adjerid, R. Balebako, L. Brandimarte, L. F. Cranor, S. Komanduri, P. G. Leon, N. Sadeh, F. Schaub, M. Sleeper *et al.*, "Nudges for privacy and security: Understanding and assisting users? choices online," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 44, 2017.

[66] H. Almuhimedi, F. Schaub, N. Sadeh, I. Adjerid, A. Acquisti, J. Gluck, L. F. Cranor, and Y. Agarwal, "Your location has been shared 5,398 times!: A field study on mobile app privacy nudging," in *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. ACM, 2015, pp. 787–796.

[67] R. Balebako, F. Schaub, I. Adjerid, A. Acquisti, and L. Cranor, "The impact of timing on the salience of smartphone app privacy notices," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2015, pp. 63–74.

[68] B. Liu, M. S. Andersen, F. Schaub, H. Almuhimedi, S. A. Zhang, N. Sadeh, Y. Agarwal, and A. Acquisti, "Follow my recommendations: A personalized privacy assistant for mobile app permissions," in *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, 2016, pp. 27–41.

[69] L. Jedrzejczyk, B. A. Price, A. K. Bandara, and B. Nuseibeh, "On the impact of real-time feedback on users' behaviour in mobile location-sharing applications," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*. ACM, 2010, p. 14.

Gian Luca Scoccia received his PhD from the Gran Sasso Science Institute, Italy, in 2019. Currently is a post-doc at the University of L'Aquila, Italy. His research focuses on software engineering, empirical software engineering, privacy and security of mobile apps, mining of software repositories, program analysis. Contact him at gianluca.scoccia@univaq.it.

Ivano Malavolta is assistant professor – Vrije Universiteit Amsterdam, The Netherlands, Department of Computer Science. His research focuses on data-driven software engineering, with a special emphasis on mobile software development, software architecture, model-driven engineering, robotics software. He is applying empirical methods to assess practices and trends in the field of software engineering. He authored several scientific articles in international journals and peer-reviewed international conferences proceedings. He is program committee member and reviewer of international conferences and journals in the software engineering field. He received a PhD in computer science from the University of L'Aquila in 2012. He is a member of ACM, IEEE, VERSEN, Amsterdam Young Academy, and Amsterdam Data Science. More information is available at http://www.ivanomalavolta.com.

Marco Autili is an associate professor – University of L'Aquila's Department of Information Engineering, Computer Science, and Mathematics. His research focuses on automated synthesis for composing distributed systems, context-oriented mobile software programming, resource-oriented analysis of mobile apps, formal specification and checking of temporal properties. Autili received a PhD in computer science from the University of L'Aquila. Contact him at marco.autili@univaq.it.

Paola Inverardi is a full professor – University of L'Aquila's Department of Information Engineering, Computer Science, and Mathematics. Her research focuses on software specification and verification of concurrent and distributed systems, deduction systems, and software architectures. Inverardi received an honorary Ph.D. in computer science from Mälardalen University. Contact her at paola.inverardi@univaq.it.

Amleto Di Salle is research fellow – University of L'Aquila's Department of Information Engineering, Computer Science, and Mathematics. His research focuses on application of software engineering methods and (practical) formalisms to the modeling, verification, analysis and automatic synthesis of component-based and service-oriented distributed systems. Di Salle received a PhD in computer science from the University of L'Aquila. More information is available at http://people.disim.univaq.it/~amletodisalle/.