A self-configuring and adaptive privacy-aware permission system for Android apps

Gian Luca Scoccia
Department of Information Engineering Computer Science and Mathematics
University of L'Aquila, Italy
gianluca.scoccia@univaq.it

Marco Autili
Department of Information Engineering Computer Science and Mathematics
University of L'Aquila, Italy
marco.autili@univaq.it

Paola Inverardi
Department of Information Engineering Computer Science and Mathematics
University of L'Aquila, Italy
paola.inverardi@univaq.it

***Abstract*—Since 2008, app stores are boosting an increasing distribution of mobile apps and, today, mobile devices keep unprecedented handy capabilities at end-users' fingertips. The price to be paid for this convenience often involves supplying and processing a certain amount of personal information, exposing end-users to novel security and privacy threats. In previous work, we proposed Android Flexible Permissions (AFP), a user-centric approach for the management of Android permissions that empowers end-users with fine-grained control over their personal data. In this paper, we extend AFP with self-configuration and self-adaptation capabilities in order to (i) ease the adoption process through the awareness of user privacy preferences, and (ii) timely adapt the permissions configuration to protect against unforeseen threats that might arise over time. Performance and accuracy of the approach implementation have been evaluated by using data collected from 46 Android users.**

***Index Terms*—Android permission system, privacy-awareness, self-configuration, self-adaptation**

I. Introduction

Since the launch of the first dedicated market in 2008, mobile apps have been experiencing a tremendous diffusion, with almost 3 million apps available today on the Google Play store, totaling over 79 billion yearly downloads [1], [2]. These apps empower end-users with unprecedented capabilities as even complex operations, such as trading stocks or recording vital health information, can be performed within pocket-reach. These increased capabilities are not without cost: mobile apps hold an unprecedented amount of personal information, exposing end-users to novel security and privacy threats [3].

Providing users with interactive systems that aim to protect their privacy has only been partially effective: end-users have been found to pay limited attention to warning dialogs and possess an inadequate comprehension of privacy and security risks [4]. Moreover, when interacting with systems that frequently require their attention, they quickly develop mental shortcuts and potentially hazardous habits [5]–[7]. Once set, privacy settings are infrequently reviewed and updated [8].

Threats to end-users' privacy and data ownership come not only from newly installed applications but can also arise from the discovery of new threats and vulnerabilities in apps that are already installed on their devices [9]. As an example, 15 different vulnerabilities have been discovered in the popular messaging app *Whatsapp*[1] since its release in 2009, according to the United States national vulnerability database [10]. These issues often linger inside apps for years, before being addressed and fixed by developers [11], [12], and today the average smartphone user has over 60 apps installed on her device [13]. Likewise, similar vulnerabilities are also found in the underlying operating system [14], seldom updated by users [15].

Aiming at addressing this issue, in a previous work [16], we proposed Android Flexible Permissions (AFP), a user-centric approach for the management of Android permissions that empowers end-users with finer grade control over their personal data. Evaluated by means of two experiments, AFP was positively received by both end-users and developers. Nonetheless, the evaluation also revealed some criticalities: (i) initial configuration of privacy choices proved burdensome to some users, and (ii) the permission system proved unable to promptly adapt and protect against new threats that might appear over time.

In this landscape, *self-configuration* and *self-adaptation* capabilities offer promising implications: the former can help in minimizing the required user interaction and ease the adoption process; the latter allows to quickly change the system configuration to protect against initially unknown threats that might arise over time. In this paper, exploiting the architecture of the existing system [16], we describe the additions we have made to introduce the aforementioned capabilities. The resulting system is thus rendered adaptive on multiple levels: first, by reasoning on observed user behavior and collected crowd-sourced information, it is able to suggest an initial configuration for privacy settings of newly installed apps; secondly, apps installed on users' devices are continuously monitored and analyzed on an existing remote server, so as to detect new vulnerabilities and promptly react through adaptation. The vulnerable app is then quarantined and its privileges to access sensitive resources are limited until app developers provide a suitable patch. Performance and accuracy of the approach implementation have been evaluated by using data collected from 46 Android users. Overall, the achieved results show that the approach is feasible and effective.

The remainder of this paper is structured as follows. Section II provides background concepts about Android permissions and AFP. Section III discusses the proposed self-

[1] www.whatsapp.com/

configuration technique and the associated architecture to enable it. Likewise, Section IV discusses the self-adaptation approach and how it is realized. Section V reports on the performed evaluation. Section VI discusses related works, and Section VII closes the paper.

II. Background

This section gives a brief account of the Android permission system in Section II-A, and a quick overview of our Android Flexible Permissions (AFP) approach [16] in Section II-B.

A. Android permissions

Android makes use of a permission system to inform users about access to sensitive information and to regulate the access to sensible APIs of the platform. Developers have to declare upfront, in the app manifest file, if and which their apps require access to security- and privacy-relevant parts of the platform and, starting with Android 6, users are prompted for confirmation when an app attempts to access a restricted part of the platform. Permissions are divided in two protection levels: *normal permissions*, that cover areas where the app needs to access resources or data with very little risk for the user's privacy (e.g., activation of the bluetooth, setting an alarm), and *dangerous permissions*, that cover areas where the app accesses data or resources that involve the user's private information (e.g., reading contacts, accessing the camera). Permissions are granted on a whole-app basis, i.e., once granted the permission is valid for the entire app, reducing control on how and when private data is accessed. Users can potentially revoke permissions granted to an app from the device system settings but the process is unwieldy and users are unlikely to do so [8].

B. Android Flexible Permissions

AFP is a user-centric approach to the management of Android permissions that empowers end-users with a finer-grained control over accesses on private or sensitive resources [16]. In detail, it allows the specification of *level-* and *feature-based permissions*. The former enables end-users to specify the granularity at which a sensible resource/data can be accessed (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) while the latter allows for the specification of 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).

The current AFP architecture involves three main components (Figure 1): the *AFP App*, the *AFP Enforcer*, and the *AFP Server*. In a typical usage scenario, the three components interact as follows: after downloading an AFP-enabled app X from the Google Play store (step 1 in Figure 1), upon its first launch (step 2), the user is automatically redirected by the embedded AFP Enforcer to the AFP App (step 3) from which the user is asked to configure her preferences for feature- and level-based permissions (step 4). Meanwhile, an integrity check is performed, generating a checksum for the app and forwarding it to the AFP Server (step 5). When the app integrity has been verified (step 6) and the configuration is finished the user is redirected to app X. During app usage, the previously specified feature- and level-based permissions are enforced by the AFP Enforcer (step 8).

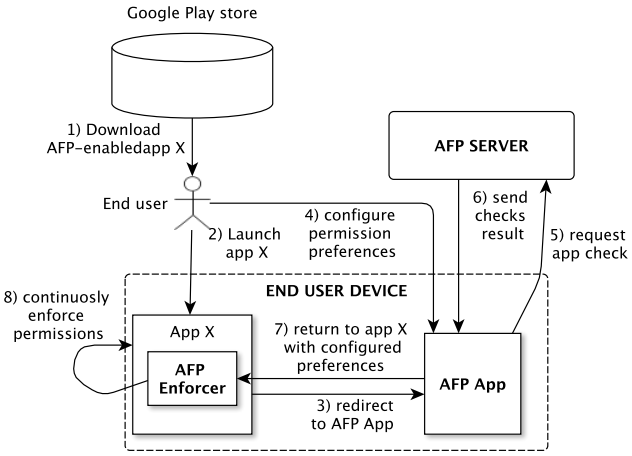

Fig. 1: Overview of the AFP approach

III. Enabling self-configuration

This section describes in what way self-configuration can take advantage from the user history (Section III-A), and how it can also be achieved in its absence (Section III-B). Then, Section III-C describes extensions made to the previous architecture of AFP to enable self-configuration, and Section III-D discusses technical aspects.

During the evaluation, end-users generally felt more secure when employing AFP and praised the added control over shared personal data provided by it [16]. However, the evaluation also revealed some weaknesses in the system: some users found the permissions' configuration process difficult and lamented the fact that it could take too long for some apps. These issues can be mitigated empowering AFP with *self-configuration* capabilities, to automatically specify its settings according to user privacy preferences. This allows to remove the initial time investment required to configure permissions, thus making the whole approach immediately usable by end-users. Towards this goal, the major challenge is devising a way to infer the user personal privacy preferences. It is clear that, even if restricting to the mobile domain, for this purpose multiple solutions may exist, each one with its own advantages and disadvantages, that have yet to be evaluated.

In the following, we discuss two different approaches for self-configuration, each applicable in distinct situations, according to the degree of available information.

A. Self-configuration from the user history

One possibility is to take advantage of the user history of past configurations. Over time, users continuously install new applications on their devices, each characterized by different attributes such as, e.g., offered functionalities, developer, and

employed APIs. It has been observed that users, when dealing with permissions, exhibit *behavioral consistency* [8], i.e., they behave consistently when faced with requests for access to a resource originating from different applications. Thus, past configurations provide a valuable source of information to enact self-configuration.

As an example, let us consider a hypothetical application called *Localgram* that allows users to take and share pictures on a social network and visualize pictures shared by others on a map. Obviously, to offer all its functionalities, the app requires permission to use the camera and the GPS. Thus, it is reasonable to assume that a user that installs the app will grant the permissions consistently with other social network and camera apps she is already using, or that she used in the past. In view of this, in the following, we describe a self-configuration procedure that mimics this user behavior.

Taking inspiration from vectorial representations commonly used in the field of information retrieval (e.g., *bag-of-words* [17]), we can construct a vectorial representation of a mobile application, suitable for automated reasoning and processing, as follows. Informally, given $A = \{a_1, \ldots, a_n\}$ the set of apps that a user has installed over time on her device, we define as $F = \{f_1, \ldots, f_m\}$ a set of heterogeneous features[2] used to describe each $a \in A$. From the above two sets, we can build a matrix M where each $M_{a_i f_j}$ indicates the number of occurrences of the feature f_j in the app a_i. Under this representation, we can express an app as a vector of features. In this vectorial form, the similarity among two apps corresponds to the distance between their corresponding vectors, which can be calculated employing one of several possible distance measures (e.g., euclidean or cosine distance [19]).

Then, defining $P(a) = \langle p_1, \ldots, p_n \rangle$ as the vector that encodes the permissions levels[3] granted to an app $a \in A$, the problem of self-configuration translates to the problem of identifying a reasonably precise vector $P(a')$ for a newly installed app a'. Values of $P(a')$ can be computed by combining (e.g., through average or statistical mode) a certain number, say k, of permissions level vectors $P(a_1), \ldots, P(a_k)$. These vectors are selected in a way that: (i) for each $p \in P(a')$ there is at least one app among the k selected that also requires p, and (ii) the distance between the k selected apps and a' is minimized. The first rule permits to infer a value for all permissions requested by the application a' (cases when this is not possible are discussed in the following); the second rule ensures that reasonably similar apps are selected as basis for the self configuration. The number k has to be empirically evaluated and existing clustering algorithm such as k-nearest neighbors [20] can be employed to perform this selection. Remind that, as mentioned in Section II-A, Android developers are required to explicitly declare the permissions employed by their app in the manifest file and hence it is possible to extract the set of permissions used by the app from it.

Example – Referring to the Localgram application previously described, we can reasonably assume that it will make use of Camera and Location APIs, invoked multiple times in the source code, in addition to referencing concepts such as "friends" and "share" in its app store description and UI templates. All of these can be considered relevant features and their occurrences can be extracted to obtain a vectorial representation of the app. Now, suppose that in the past the user had installed the popular Instagram app (another social network app that focuses on photo sharing), we can also reasonably assume that its vectorial representation is similar to the one of Localgram, as the two apps are similar in concept and functionalities. Hence, the distance between corresponding vectors is narrow and Instagram will be chosen as one of the k neighbors of Localgram. Localgram will be then self-configured with the same settings for those permissions employed by both apps (i.e., camera and location).

B. Self-configuration without the user history

In some cases, relying on the user configuration history is not possible. This might happen for users that have started using AFP only recently, or those that install new apps on their device infrequently. In such cases, their configuration history does not provide enough information to accurately infer their preferences. We introduce a maximum distance threshold t in the self-configuration process described in Section III-A. Basically, given a newly installed app a', the system starts the self-configuration process to calculate the values of $P(a')$. If the distance between a' and some of its k neighbors is found to be greater than t, it means that they differ too much in functionalities and characteristics from the app a' and, hence, should not be used as a basis to deduct the permission levels for the app a'. Depending on the amount of available history, two different cases are possible:

i) the user history is *partially incomplete*, i.e., there is a limited number of $p \in P(a')$ for which there is no neighbor within the distance t that also makes use of p. In other words, only a few of the k-nearest neighbors identified during the self-configuration fall outside of the distance threshold t.
ii) the user history is *mostly incomplete*, i.e., for the majority of $p \in P(a')$ there is no neighbor within distance t that also makes use of p. In other words, most of the k-nearest neighbors identified during the self-configuration fall outside of the distance threshold t.

A graphical representation of both cases is given in Figure 2. Here, considering $t = t_1$, we get that only d_1, the distance of a_1 from a', exceeds the distance threshold t and hence we can consider the user history as partially incomplete. Instead, setting $t = t_2$, most of the distances fall outside of distance threshold t, with the sole exception being d_4, distance of a_4 from a', and we fall in the case of mostly incomplete history.

We address these cases resorting to *crowdsourcing* of permissions configurations, i.e., combining information collected

[2]The term feature is used in its machine learning connotation, denoting an individual measurable property or characteristic of a phenomenon being observed [18].

[3]Note that this vector is not binary since, in AFP, permissions can have multiple intermediate levels.

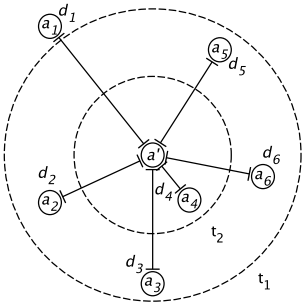

Fig. 2: Graphical representation of incomplete user history ($t = t_1$ partially incomplete, $t = t_2$ mostly incomplete)

from multiple users, as explained in the following. In the case of partially incomplete history, crowdsourcing can be leveraged to augment the available information, replacing those applications outside of the distance threshold with alternatives collected from other AFP users. In this way, the self-configuration process can still be carried out to completion while retaining in consideration, for as much as possible, the user preferences. In the case of mostly incomplete history, available information is too shallow to perform the self-configuration process in a way that is tailored to user preferences. Hence, as a fallback strategy, values for $P(a')$ are averaged from choices of other users that have accumulated considerable experience in using the system and demonstrated to be mindful of their privacy. Clearly, to enable both strategies it is necessary to collect information about individual users, profiling and dividing them into clusters, depending on their privacy attitudes. Then, from within each cluster, information can be shared among users. For our purpose, permissions configurations expressed by users for their apps can be collected and analyzed in the AFP server.

Example – Let us assume that a user for which history is partial incomplete installs the Localgram app, described in Section III-A. During the self-configuration process, no neighbor is found that also makes use of the device camera, so the system proceeds to request, from other users in the same cluster, the features of an app similar to the one being installed that also makes use of the device camera (e.g., the Instagram app). These are used in place of the missing neighbours and the self-configuration can be brought to completion successfully. Instead, if the user is a newcomer and his configuration history is mostly incomplete, configurations for the permissions employed by the Localgram is generated employing the settings of expert users that also have installed the same app.

C. Self-configuration architecture

The self-configuration process is enacted through the interaction of the AFP App and the AFP Server. To carry out the process, new components have been added to both the client-side and the server-side. Additions to the former are the *AFP Monitor*, the *AFP Actuator*, and the *AFP Profiler*; the latter has been augmented with the *AFP Collector*, the *AFP Feature Extractor*, the *AFP App Database*, the *AFP App Configuration Generator*, the *AFP Profile Database*, and the *AFP Profile Clusterer*. Figure 3 shows how the whole self-configuration process is split among the newly added components and, in the following, we describe the interaction flow among these components.

Whenever the user installs a new application X (step 1 in Figure 3), the newly installed package is detected by the AFP Monitor (step 2). The latter, in turn, transmits the app id (i.e., its package name) to the AFP Collector that resides on the AFP Server. This is done to offload expensive computations to the AFP Server, in order not to overload the user's mobile device, both processing- and energy-constrained. Likewise, only the id of app X is transmitted to the server, rather than the whole binary, in order to limit the consumption of the user's network data. At this point, the AFP Collector searches for app X on multiple app stores and retrieves its binary package and other artifacts related to it (e.g., app store description, user reviews, app privacy policy) (step 4).

Afterward, the artifacts are handed over to the AFP Feature Extractor, which is in charge of collecting the features required for the self-configuration process (step 5). Since the features are heterogeneous in nature, several specialized extraction engines run in parallel, each on a different artifact related to the app. Note that, in addition to speed up the extraction process, specialized engines require a lower maintenance effort as compared to a monolithic extractor. Moreover, during the system lifetime, more features can be classified as relevant, hence allowing for evolving the system through the modular inclusion of more extraction engines.

The extracted features are stored in the AFP App Database for future use (step 6) where other similar apps used by the user (e.g., the neighbors resulting from the process described in Section III-A) are also identified (step 7). Features of these apps, plus those of app X, are handed to the AFP Configuration Generator (step 8), which is in charge of generating the permissions configuration $P(X)$, also leveraging the configurations of similar apps retrieved from the AFP Profile Database (step 9a). If necessary, crowdsourced information required to actuate the alternative strategies explained in Section III-B is also retrieved (step 9b).

The permission configuration is then transmitted back to the client-side AFP Actuator (step 10), which is in charge of applying it to app X (step 11). The new configuration is also used by the AFP Profiler to update the user profile (step 12), which is in turn transmitted to the server (step 13) and memorized in the AFP Profile Database (step 14). Periodically, users profiles are assigned to clusters (step 14), to group together users with akin privacy preferences and identify the more privacy-aware users. The user profile is updated (repeating steps 13 and 14), both locally and on the server-side, whenever the user chooses to change the permissions of app X (step 15).

D. Technical aspects

To effectively suggest precise configurations for the newly installed apps, the described approach requires the availability

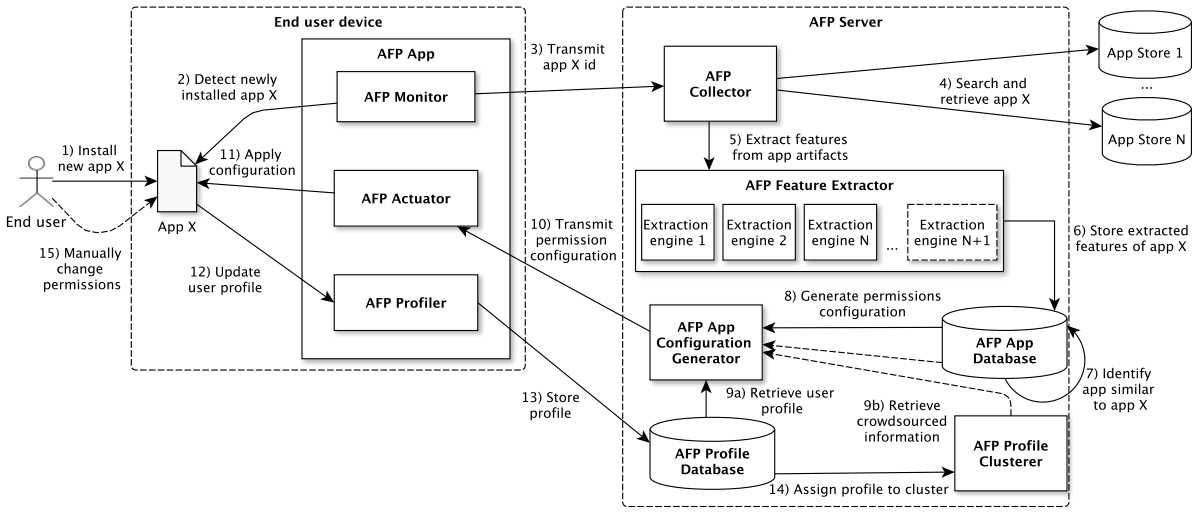

Fig. 3: Self-configuration procedure (dotted arrows are optional steps)

of many pieces of information about the new and past installed apps, so that their similarity (i.e., the distance) can be properly evaluated. It thus becomes important that leveraged features are heterogeneous, descriptive and, to preserve scalability of the approach, computationally inexpensive to extract. For this purpose, we plan on prioritizing features that can be collected without the need of executing the app or performing complex analyses. The features can be extracted from the app itself and from other artifacts related to it, e.g., the app store description, the app privacy policy and its user reviews. Feature selection algorithms [21] can be used to identify those features that are more relevant for our goals. Moreover, in the AFP Server, we make use of lightweight virtualized containers [22] to allow for greater scalability and to easily accommodate additional extraction engines. In Section V we describe our implementation of the self-configuration approach and provide an evaluation of its accuracy and performance.

An important aspect to take into consideration is that users' privacy preferences can change over time and thus apps that have been configured in a remote past may not be representative of the user's current preferences. It is therefore important to prioritize those apps that have been installed more recently during the self-configuration. This can be done taking into consideration the app installation date as one of the features that are used to compute distance among apps, in a way that more recently installed apps are considered closer.

In order to deal with scenarios for which the user history is only partially available, the two strategies described in Section III-B require the collection of the permissions levels granted to a large number of Android apps. These are sensitive information, as these configurations express the users' habits towards privacy and security settings. Hence, this data will be anonymized during collection and it will be collected only for those users who expressly consent to it. Additionally, it will be processed and aggregated by use of privacy-preserving algorithms (e.g., *differential privacy* techniques [23]) that, while preserving patterns within the data, withhold information about individual users.

The Android platform raises an *Intent* [24] (i.e., a broadcast message that apps can listen for and take action in response to) whenever a new app is installed or updated on the device. Hence, the AFP Monitor does not need to be continuously active in the background as, registering the proper Intent, the operating system itself awakens it when needed. This mechanism helps in reducing the logical complexity of the monitoring process, and the energy consumption due to its execution.

IV. Enabling self-adaptation

This section describes how permissions are adapted by the enhanced AFP (Section IV-A), and how its architecture has been extended to enable self-adaptation (Section IV-B). Technical aspects are then discussed in Section IV-C.

Assisting users in the initial configuration steps is not enough to ensure the proper protection of their privacy and data ownership, as new threats continuously emerge over time. Indeed, not only users constantly install new apps on their devices but also novel threats might be discovered in apps already installed on the device, as a consequence of software updates or the discovery of novel security vulnerabilities. To address these problems, we introduce *self-adaptation* capabilities in AFP, in order to promptly identify and adapt to novel threats that might appear over time.

A. Self-adaptation of permissions

As defined in Section III-A, the vector $P(a) = \langle p_1, \ldots, p_n \rangle$ encodes the permissions levels granted to an app $a \in A$ (remember that this vector is not binary as, in AFP, permissions can also have intermediate levels). Given a permission level p_i, employed by a, we denote with $T(a, p_i)$ a newly discovered threat or vulnerability in a that involves the permission p_i. As described in the next section, the AFP analyzer makes use of different analysis engines to detect the vulnerability and assess

that its impact can be mitigated by reducing the permission level p_i by δ so as to permit a more restrictive access to the affected sensitive resources. The goal of the self-adaptation process is then to identify a new vector $P'(a)$ such that:

i) for each permission level $p_i \in P(a)$, the corresponding level $p'_i \in P'(a)$ is such that p'_i is less than or equal to the permission level p_i, hence $p'_i \leq p_i - \delta$;
ii) $P'(a)$ is a maximizing assignment (i.e., less restrictive possible) among all possible ones that complies with i).

The above can straightforwardly be translated into an *integer programming* problem [25], whose constraints and objective function "encode and force" the two clauses i) and ii). The positive aspect here is that, in integer programming, when the number of variables is fixed and limited, the problem can be solved efficiently in polynomial time by means of different algorithms (e.g., branch-and-bound algorithms).

Example – Let us assume that a user has installed the LocalGram app described in the previous example of Section III-A. A few weeks after, a new updated version of the app is made available by its developer. Unfortunately, the newer version of the app exploits the storage permission to covertly collect pictures that the user has used in other social network apps (such as Instagram). This unwanted behavior is detected by AFP and the self-adaptation process is enacted: the app is automatically quarantined and it is now allowed to access only a restricted set of selected folders in the device storage.

B. Self-adaptation architecture

The self-adaptation capabilities are realized through interaction between the AFP App and the AFP Server by leveraging not only the components specifically introduced for self-adaptation, but also the ones introduced to enact the self-configuration previously described in Section III-C: the client-side *AFP Monitor* and *AFP Actuator*, along with the server-side *AFP Analyzer*, *AFP Collector*, *AFP Console*, and *AFP Threat Database*. Hereafter, we will explain how these components interact referring to Figure 4.

On the client side, the AFP Monitor identifies newly installed or updated applications (step 1) and transmits their identifiers (i.e., the Android package names) and version numbers to the AFP Server (step 2). As for the self-configuration, this offloading is done in order to avoid expensive computations on the users' device and only the app id is transmitted to the server to reduce the consumption of network data. Here, the package is retrieved from the AFP Collector, which queries for it on multiple app stores and downloads the matching application from one of them (step 3). The application is given as input to the AFP Analyzer, which in turn proceeds to scrutinize the app for vulnerabilities, running a series of analyses in parallel (step 4). As in the case of self-configuration, novel analysis engines can be added to the AFP Analyzer during the system lifetime in order to evolve the threats identification ability of AFP and add new analysis techniques that might appear over-time.

Whenever a new threat or security vulnerability is detected, it is inserted in the AFP Threat database (step 5) and a quarantine procedure is initiated: the AFP Server notifies the AFP Actuator residing on the affected user device (step 6), which in turn limits the granted permission levels and notifies the user (step 7). The goal is to minimize the impact on the end-user privacy. Afterwards, the system audits the updates of the affected app (step 8), until one that solves the issue is released (step 9). At this point, the AFP Administrator is promptly notified (step 10) and, if she concurs that the vulnerability is no longer present in the updated version, she can lift the quarantine of app X. This will result in the AFP Actuator being notified (step 11), and the user being invited to update the app (step 12). Afterwards the restrictions on updated versions of the app will be automatically removed (step 13). Note that, in order to also take into consideration threats or vulnerabilities newly discovered by third-parties, the quarantine process can also be started by the AFP administrator, who can manually add those to the AFP Threat Database from the administrator console (step 14), thus triggering the self-adaptation process.

C. Technical aspects

The Android platform raises an Intent [24], i.e., a broadcast message, not only when a new application is installed, but also when an already installed application is updated. Similarly to what happens for self-configuration of newly installed apps (see Section III-D), the AFP Monitor can monitor updated apps registering for the proper Intent. This significantly reduces the energy consumption for the AFP Monitor execution.

Since the threats to which users are exposed are constantly evolving, with new vulnerabilities and exploits revealed every day, the ability of the analyzer to accommodate varied and constantly updated analysis engines is of crucial importance. To this end, for the AFP Analyzer, we adopt a modular architecture in which each analysis engine is self-contained in a lightweight virtualized container [22]. Each container encapsulates the environment suitable for the execution of a distinct technique, thus enabling the combination of very different analyses within the AFP Analyzer, taking advantage of both static and dynamic techniques offered through the extensive available literature [3], [26], as well as off-the-shelf tools [27]. Moreover, lightweight virtualization technologies allow for greater scalability, by enabling the addition and the removal of containers on-the-fly, thus permitting the prompt reconfiguration of the system to accommodate additional analyzers or to cope with periods in which there is a greater number of analysis requests.

It is also vital to quickly and efficiently solve the integer programming problem (Section III-A), in order to promptly transit to the new permission settings and deal with the identified threat. For this purpose, we leverage the consolidated off-the-shelf Gurobi [28] solver. It has been developed with a keen eye on performance and, as such, is able to solve the adaptation problem at hand in a timely manner [29].

V. Evaluation

For evaluation purposes, we engineered a prototype implementation of the self-configuration algorithm proposed in

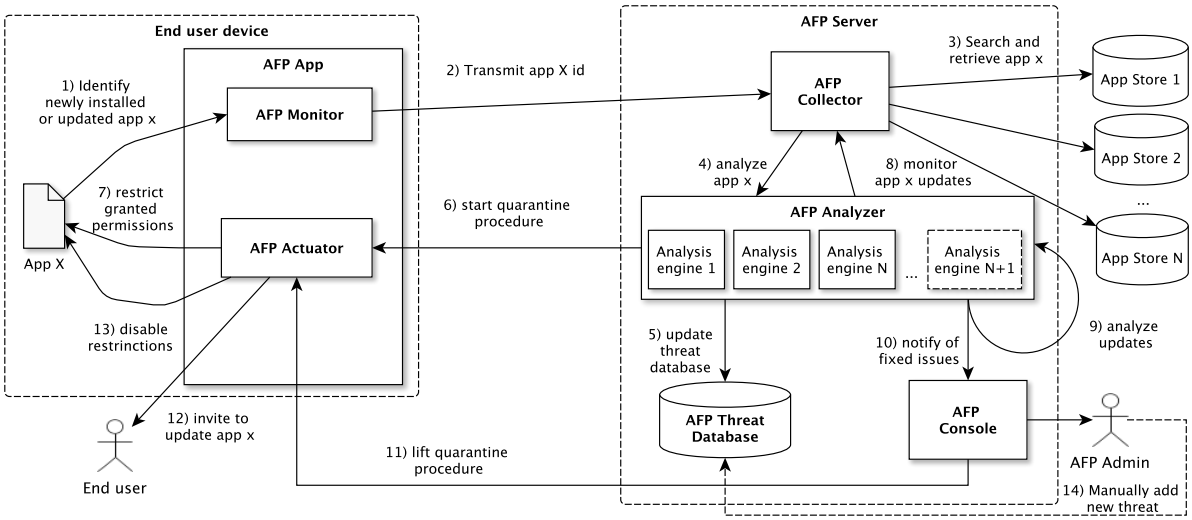

Fig. 4: Self-adaptation procedure (dotted arrows are optional steps)

Section III-A. In our implementation, we make use of a two extraction engines: the first extracts app features from the apps' own descriptions, available on the Google Play Store (a method that has proven to be effective in previous works [30]), while the latter collects permissions information declared in each app own manifest file. Nonetheless, the proposed approach remains general and, in the future, we plan on implementing additional extraction engines that can extract features from other sources (e.g., collected through static app analysis). Specifically, to mine features from app descriptions we resort to *topic modeling* using the *Latent Dirichlet Allocation* (LDA) [31] algorithm. For LDA a "topic" consists of a cluster of words that frequently occur together. For instance, when analyzing a corpus of app descriptions, LDA identifies a topic composed by words such as "music", "player", "audio" and "song" (which frequently appear in the description of music streaming apps), and another composed by the words "bank", "balance", "transaction", and "credit" (which frequently appear in the descriptions of banking applications). Each application would then be assigned one or more topics depending on the words occurring in their description. We decided to leverage the assigned topics as features for the self-configuration algorithm since they convey the app major functionalities. We used a corpus of 17.164 app descriptions, collected in a previous study [32], as training data for the LDA algorithm (a process that has to be executed only once on the AFP Server). Prior to the training, we filter out descriptions written in a non-english language and we apply standard preprocessing steps in the form of stopwords-removal and lemmatization [17]. We use the implementation of LDA included in the `scikit-learn`[4] Python package (version 0.22.2), setting the number of topics to 100, and we rely on grid-search (i.e., exhaustive search) in order to tune the algorithm hyperparameters. Furthermore, we consider an app related to a topic only if its probability for that topic (estimated by LDA) is at least 5%, discarding the rest.

The features produced by the extraction engines are then used as input for the configuration generator. In our implementation, we rely on the *K-Nearest Neighbor Classification* [33] (K-NN) algorithm to realize the configuration generator. In simple terms, when queried for a permission configuration for an app X, the K-NN algorithms selects among apps already installed the K most similar to X and produces the configuration leveraging the ones of the K neighbors. The K-NN algorithm is executed once for each permission of X that needs to be configured and, during each execution, the search of the K neighbors is restricted to the apps that employ that permission. A majority vote rule among the K neighbors is employed to determine the value to be assigned to X. As above, we leverage the K-NN implementation included in the `scikit-learn`[4] Python package, choosing the euclidean distance as distance formula. As described in the following, we experimented with multiple values of K.

We perform the evaluation conducting an experiment focused on two key aspects of our approach: (i) the *performance* of the self-configuration engine, and (ii) the *accuracy* of the permission configurations produced by it. Indeed, for the end-user to experience tangible benefits, both must be satisfactory. To evaluate the first, we measure the time in seconds needed to generate a configuration. To evaluate the second, we measure how close the generated configurations are to those given by users, employing the precision, recall and accuracy metrics [18]. For the experiment, we employ the dataset provided by Andriotis et al. [8], that collected a snapshot of the apps installed on the device of 49 Android users, complete with the permissions assigned to each app (under the canonical Android permission system described in Section II-A). Since app descriptions are not included in the dataset, we have collected and integrated them. However, not all apps in the dataset are still available on the Google Play Store (this can happen if the developers decided to remove

[4]https://scikit-learn.org/stable/

the app from the store or if Google decided to remove the app for violation of some publishing policies). Therefore, we have excluded those apps for which it was not possible to retrieve the description and the snapshot of 3 users who, following this filtering, were found to have less than 15 surviving apps. The remaining 46 snapshots have a number of apps ranging from a minimum of 16 to a maximum of 136, with an average of 45.91 and 25.9 standard deviation. The total amount of individual apps in the dataset is 1380.

We conducted the experiment as follows. For each of the 46 users, we generated a configuration for each of the installed apps following a leave-one-out cross validation scheme [18]. Under this scheme, in turn, the configuration for one of the installed app was generated employing the data provided by the configurations of the other apps. The generated configuration was then compared to the user defined configuration, which is considered the ground truth. The process was repeated exhaustively, until configurations were generated for all the installed apps. Since there is only a binary ground truth (i.e., in the dataset permissions are either *granted* or *denied*), we also limited the configurations generated by our system to binary values, so to allow for an appropriate comparison. We chose the leave-one-out cross validation scheme as, we believe, it closely reflects the typical application scenario of the proposed self-configuration approach, which comes into operation when the user (who already has a pool of apps installed on his device) chooses to install a new one. The whole experiment was performed on a 2014 MacBook Pro running macOS 10.14.6 and equipped with 8GB RAM and an 2.6 GHz Intel Core i5 Processor.

The self-configuration performances observed during the experiment are provided by the boxplots in Figure 5. It can be observed that the preprocessing steps are those which contribute to the execution time the most, even if only in the worst cases the time required surpasses 100ms. Instead, the median time for the extraction of features through topic modeling is of only a few milliseconds. We can also observe that as the number of neighbors K increases, the time required for the *K-nn* algorithm also slightly increases. However, with few exceptions, it is an order of magnitude lower than the time required for the preprocessing steps. Therefore, increasing the number of neighbors K does not have a significant impact on the overall performance, which is acceptable even in the worst case with a required time of about 500ms. We consider this waiting time acceptable since the self-configuration procedure is performed only when a new app is installed.

Table I summarizes the observed values for precision, recall, and accuracy. The best results are achieved in the configuration in which the number of neighbors K is set to 3, which resulted in a mean precision of 0.767 and a mean recall value of 0.703. Accuracy consistently scores a mean value above 0.9 for all tested configurations. Increasing the number of neighbors leads to a slight decline in all metrics, likely due to the fact that less similar apps are taken into account when increasing the number of neighbors. However, we observed that achieved values are lower for the permissions related

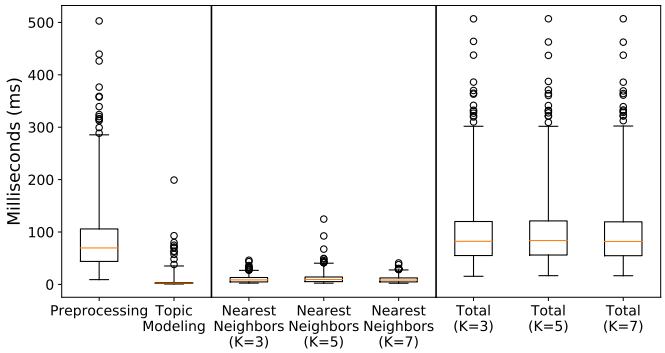

Fig. 5: Performance of self-configuration

to the device sensors and calendar. We observed that these permissions are the ones less commonly used by apps in the dataset (with only 70 apps employing the former and 6 employing the latter). Therefore, we believe that the accuracy for these permissions has been negatively affected by the relative lack of data. Hence, in Table I, we also report the values achieved excluding the configurations relating to the calendar and sensors permissions. A marked improvement can be observed for all metrics, with a mean precision value of 0.848 and a mean recall value of 0.768 when K is fixed to 3. This provides evidence of the importance of crowdsourcing to increase the data available to self-configure. Overall, results are encouraging and we believe they can further improve by expanding the set of features adopted.

Our experiment was not performed on mobile devices and, although we have used consumer-grade hardware, the performance may be lower on these devices. In total, the dataset used for validation contains 46 users. These are a very limited number when compared to the number of Android users worldwide. Therefore, there is a risk that our results may not generalize to the broader population. Finally, we did not involve Android users in person in the evaluation of the approach. To address these threats and limitations, we plan to conduct a more in-depth experiment in the future, directly involving Android users.

		All permissions				Excluding Sensors and Calendar permissions			
K	**Metric**	**Mean**	**Min**	**Max**	**Std Dev**	**Mean**	**Min**	**Max**	**Std Dev**
3	P	0.767	0.640	0.878	0.062	0.848	0.681	0.995	0.079
	R	0.703	0.497	0.819	0.058	0.768	0.504	0.912	0.074
	A	0.941	0.888	0.975	0.023	0.933	0.881	0.973	0.026
5	P	0.738	0.508	0.875	0.084	0.813	0.514	0.989	0.102
	R	0.678	0.458	0.777	0.080	0.739	0.449	0.860	0.098
	A	0.927	0.840	0.967	0.030	0.919	0.830	0.967	0.034
7	P	0.702	0.412	0.868	0.099	0.771	0.407	0.986	0.125
	R	0.642	0.343	0.765	0.093	0.692	0.345	0.841	0.113
	A	0.914	0.806	0.963	0.038	0.905	0.786	0.958	0.041

TABLE I: Precision, recall, and accuracy of self-configurations over all users (P = Precision, R = Recall, A = Accuracy)

VI. Related work

In the literature, a growing body of work is focusing on the application of self-adaptation techniques to improve mobile apps. An overview is provided by Grua et al. that conducted a systematic literature review of approaches that apply self-adaptability in the context of mobile applications [34]. Importantly, they note the absence of approaches that employ these techniques to improve system security and user privacy. Our work positions itself within this gap and, to the best of our knowledge, it is the first attempt towards addressing these challenges.

Several works have investigated the usage of machine learning techniques to automatically configure permissions. Differently from us, these works do not take into account the fact that users might express similar permission preferences across multiple apps that offer similar functionalities, and limit the inference of user preferences to the app currently being executed. SmarPer [35] 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 [36], to assess current performance and practical limitations in actual implementations. Autoper [37], rather than using run-time information, analyzes application descriptions to assess whether a permission is needed by an application.

Multiple studies have investigated the introduction of finer-grade controls over permissions using varied strategies [38]–[42]. Mockdroid [38], TISSA [39] and SHAMDROID [40] allow users to restrict access to a resource by providing mocked information in place of real one whenever access to the resource is attempted. This strategy allows the majority of applications to continue execution, although at the expense of potential usefulness. Apex [41] 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. Dr. Android [42] allows for the specification of finer-grained variants of Android permissions, by grouping existing Android permission into a taxonomy of four major groups, each of which admits some common strategies for deriving sub-permissions. None of the above assists the user in the specification of the finer-grade permissions they introduce. In contrast, our work aims at easing the knowledge and the effort required from end-users, by means of self-configuration.

Multiple works have proposed monitoring techniques for the identification of mobile malware [43]–[46]. Schmidt [43] and colleagues have first investigated the usefulness of such techniques for the identification of mobile malware, relying on the offloading of computation to a remote server to cope with the limited capabilities and hardware of mobile devices. Kirin [44] performs lightweight security analysis of Android applications to identify malware at install time, relying on a rule-based approach. Burguera et al. [45] analyze app execution traces, crowdsourced from multiple users, as a means for detecting Android malware. Andromaly [46] continuously monitors various features and events obtained from the mobile device. Then, anomaly detection is performed by means of machine learning to classify behavior of apps as benign or malicious. Our work goes beyond the bare identification of malware in that we propose a self-adaptation strategy to minimize the impact of assessed threats on end-users privacy. Mobile device management [47] suites provide enterprise solutions that assist expert technicians in the task of mobile fleet management. Similarly to our approach, these suites allow for application vulnerability detection and app quarantining in a centralized manner. However, in contrast to our work, end-users are not directly involved in app quarantining decisions.

VII. Conclusion and future work

In this paper, we enhance our previous work Android Flexible Permissions (AFP) [16] with *self-configuration* and *self-adaptation* capabilities. AFP is a user-centric approach for the management of Android permissions that empowers end-users with finer-grade control over their personal data. Exploiting the existing AFP architecture, we describe the components we added to enable self-configuration and self-adaptation capabilities and how they interact, while discussing technical aspects. The new AFP system rendered adaptive on multiple levels: (i) by reasoning on observed user behavior and collected crowdsourced information, the added self-configuration capability now permits to minimize the required user interaction and ease the adoption process; (ii) by exploiting a remote server to continuously monitor and analyze the installed apps, new vulnerabilities are promptly detected so that self-adaptation can quickly change the system configuration to protect against initially unknown threats that might arise over time.

In previous work [16], AFP was evaluated by means of two experiments, and it was positively received by both end-users and developers. In this paper, we conducted an additional experiment to evaluate the performance and accuracy of the newly proposed self-configuration engine. Specifically, the criticalities we alleviated concern (i) the initial configuration of privacy choices that proved burdensome to some users, and (ii) the rigidity to promptly adapt and protect against new threats that might appear over time.

As future work, we plan to extensively evaluate the enhanced version of AFP presented in this paper. Meanwhile, within the scope of the Exosoul [48] project, we are investigating how end-users privacy and ethical preferences can be collected and encoded into a declarative representation that enables automated processing and reasoning.

References

[1] Statista, "Number of available applications in the google play store from december 2009 to december 2017," 2017.

[2] S. Tower's, "New store intelligence data digest q4 2018," 2018.

[3] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[4] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. of the 8th Symposium on Usable Privacy and Security*, 2012, p. 3.

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

[6] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness," in *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 257–272.

[7] S. Motiee, K. Hawkey, and K. Beznosov, "Do windows users follow the principle of least privilege?: investigating user account control practices," in *Proc. of the Sixth Symposium on Usable Privacy and Security*. ACM, 2010, p. 1.

[8] 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.

[9] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System," in *28th Security Symposium (USENIX)*, 2019, pp. 603–620.

[10] N. I. of Standards and Technology. Whatsapp known vulnerabilities. [Online]. Available: https://nvd.nist.gov/vuln/search/results?query=whatsapp

[11] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on Android-related vulnerabilities," in *14th Int. Conf. on Mining Software Repositories (MSR)*, 2017, pp. 2–13.

[12] G. L. Scoccia, A. Peruma, V. Pujols, I. Malavolta, and D. E. Krutz, "Permission Issues in Open-source Android Apps: an Exploratory Study," in *18th Int. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, 2019.

[13] A. Annie, "App annie 2016 retrospective: Mobile's continued momentum," *San Francisco, CA*, 2017.

[14] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities," *arXiv preprint arXiv:1905.09352*, 2019.

[15] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Int. Conf. on Software Maintenance*, 2013, pp. 70–79.

[16] G. L. Scoccia, I. Malavolta, M. Autili, A. Di Salle, and P. Inverardi, "Enhancing Trustability of Android Applications via User-Centric Flexible Permissions," *IEEE Transactions on Software Engineering*, 2019.

[17] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[18] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[19] A. Singhal *et al.*, "Modern information retrieval: A brief overview," *IEEE Data Eng. Bull.*, vol. 24, no. 4, pp. 35–43, 2001.

[20] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[21] H. Liu and H. Motoda, *Computational methods of feature selection*. CRC Press, 2007.

[22] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[23] C. Dwork, A. Roth *et al.*, "The algorithmic foundations of differential privacy." *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3-4, pp. 211–407, 2014.

[24] A. Developers. (2019) Intent reference guide. [Online]. Available: https://developer.android.com/reference/android/content/Intent.html

[25] L. A. Wolsey and G. L. Nemhauser, *Integer and combinatorial optimization*. John Wiley & Sons, 1999, vol. 55.

[26] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[27] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www.virustotal.com/en*, 2020.

[28] "Gurobi optimizer reference manual, 2015," 2014.

[29] B. Meindl and M. Templ, "Analysis of commercial and free and open source solvers for linear optimization problems," *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, vol. 20, 2012.

[30] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE transactions on software engineering*, vol. 43, no. 9, pp. 817–847, 2016.

[31] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[32] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi, "An investigation into android run-time permissions from the end users' perspective," in *Proc. of the 5th Int. Conf. on Mobile Software Engineering and Systems*, 2018, pp. 45–55.

[33] S. A. Dudani, "The distance-weighted k-nearest-neighbor rule," *IEEE Transactions on Systems, Man, and Cybernetics*, no. 4, pp. 325–327, 1976.

[34] E. M. Grua, I. Malavolta, and P. Lago, "Self-adaptation in mobile apps: a systematic literature study," in *14th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2019, pp. 51–62.

[35] 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 *IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 1058–1076.

[36] 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 *Proc. of the Conf. on Human Factors in Computing Systems*, 2018, p. 268.

[37] H. Gao, C. Guo, Y. Wu, N. Dong, X. Hou, S. Xu, and J. Xu, "Autoper: Automatic recommender for runtime-permission in android applications," in *43rd Annual Computer Software and Applications Conf. (COMPSAC)*, vol. 1, 2019, pp. 107–116.

[38] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proc. of the 12th workshop on mobile computing systems and applications*, 2011, pp. 49–54.

[39] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Int. Conf. on Trust and trustworthy computing*, 2011, pp. 93–107.

[40] L. Brutschy, P. Ferrara, O. Tripp, and M. Pistoia, "Shamdroid: gracefully degrading functionality in the presence of limited resource access," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 316–331, 2015.

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

[42] 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 *Proc. of the second ACM workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012, pp. 3–14.

[43] A.-D. Schmidt, F. Peters, F. Lamour, C. Scheel, S. A. Çamtepe, and Ş. Albayrak, "Monitoring smartphones for anomaly detection," *Mobile Networks and Applications*, vol. 14, no. 1, pp. 92–106, 2009.

[44] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. of the 16th ACM Conf. on Computer and communications security*, 2009, pp. 235–245.

[45] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proc. of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011, pp. 15–26.

[46] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[47] L. Liu, R. Moulic, and D. Shea, "Cloud service portal for mobile device management," in *2010 IEEE 7th International Conference on E-Business Engineering*. IEEE, 2010, pp. 474–478.

[48] M. Autili, D. Di Ruscio, P. Inverardi, P. Pelliccione, and M. Tivoli, "A software exoskeleton to protect and support citizen's ethics and privacy in the digital world," *IEEE Access*, vol. 7, pp. 62 011–62 021, 2019.