How to Deal with Third Party Apps in a Privacy System —The PMP Gatekeeper—

Christoph Stach†
University of Stuttgart, Institute for Parallel and Distributed Systems
Universitätsstraße 38, 70569 Stuttgart, Germany
Email: Christoph.Stach@ipvs.uni-stuttgart.de

***Abstract*—Nowadays, mobile devices collect a lot of private information. Therefore every vendor of a mobile platform has to provide a sufficient mechanism to secure this data. Android pursues a strategy to pass full control (and thus full responsibility) over any private data to the user. However, the Android Permission System is not sufficient for that purpose. Various third party approaches try to enhance the Android privacy policy model. Nevertheless, these approaches have to solve the problem of how to deal with *Legacy Apps*, i. e., apps that do not collaborate with an enhanced privacy policy model.**

In this paper, we analyze various alternative privacy systems and discuss different approaches of how to deal with Legacy Apps. Based on our findings, we introduce the so-called *PMP Gatekeeper*, a best of breed approach dealing with Legacy Apps for the Privacy Management Platform (*PMP*). The PMP Gatekeeper classifies apps and deals with each class appropriately. So the user can adjust privacy settings for every kind of app. With our prototype we show, that the PMP in combination with the PMP Gatekeeper becomes a holistic privacy system. Although our prototype is for Android, our realization approach can be applied to other application platforms in order to offer a satisfying privacy system.

***Index Terms*—Android; Privacy Systems; Legacy Apps.**

I. Introduction

The times in which mobile phones are used for calls or short text messages only are long gone. The so-called smart phones are capable to provide almost any service which is known from a desktop PC, e. g., making appointments, managing contacts, reading and writing emails, or accessing the Internet. Additionally, due to the advanced technical features of these devices, their range of services goes even beyond the one of a PC. Thus, it is possible to use navigation services via GPS or to make cashless payments via NFC. Moreover, almost countless third party applications (*apps*) are available via online software repositories (the so-called *app stores*). Due to the thereby increasing amount of personal data stored on these devices, such apps are a potential privacy violation. Since the scope of available apps is broad—ranging from carpooling apps (e. g., *vHike* [1], [2]) to *mHealth* services (e. g., *Candy Castle* [3])—also the range of stored data is manifold. Therefore, especially apps collecting and processing sensitive data (e. g., health data) should be monitored with extreme care by the data owner in order to ensure that these data are not misused.

† This work was supported by a Google Research Award.

However, the users themselves cannot check what is done with their data and with whom it is shared. For this purpose, the users depend on the support of their mobile platform vendor. Since the mobile platform market is dominated today by Google's *Android* with a market share of about 85 %, we only consider this OS in the following. In Android the user is in total control over his or her private data—and therewith full responsibility over this data is transferred to him or her at the same time. Therefore, Android operates according to the *Principle of Least Privilege*, i. e., every app can access only data and system resources that are necessary for its legitimate purpose. To get access to a protected content or hardware function (e. g., the current position via GPS or the usage of the camera) an app requires a so-called *Permission*. However, the user has to grant either all of an app's Permissions or the app cannot be installed at all. To make matters worse, most of the apps request a large number of Permissions—often even totally unnecessary ones—so the users' privacy is increasingly at risk [4]. Therefore, a lot of different third party approaches are developed to improve the Permission system. However, as apps support only the original Permission system with its policy model and are not adapted to collaborate with enhanced models, one key problem of these approaches is the question of how to deal with these third party apps (labeled as *Legacy Apps* in the following). Basically, there are two distinct strategies each with its own benefits and drawbacks (see Section III).

We introduce with the *PMP Gatekeeper* an extension to the **P**rivacy **M**anagement **P**latform (*PMP*) [5], making the PMP to the first privacy system supporting both Legacy Apps strategies. Therefore, the following three requirements are addressed:

(1) The PMP Gatekeeper allows users to specify access rights which should be granted to an app.
(2) Virtually any kind of app can be monitored and regulated by the PMP thanks to the PMP Gatekeeper.
(3) As system functions must not be interfered by a privacy system, the PMP Gatekeeper is able to identify system services and it gives them any required access right automatically.

Especially the third issue is an important one due to stability and performance reasons.

The remainder of this paper is as follows: In Section II we give a comprehensive overview of currently existing privacy systems and differentiate the PMP from state of the art approaches. On the basis of this overview, we deduce two different strategies of how to deal with Legacy Apps in Section III (namely the *extension of the privacy system* in Section III-A and the usage of *Inline Reference Monitoring* in Section III-B). Subsequently, we introduce the PMP Gatekeeper in Section IV and describe in detail how the three key issues are realized by the PMP Gatekeeper. Section V discusses some implementation insights of the PMP Gatekeeper. Finally, Section VI assesses the approach before Section VII provides a short summary of this paper.

II. Related Work

A first approach towards a better understanding for privacy threats is to give the user a profound characterization of the Permissions an app has and how dangerous certain Permission combinations are [6]. It is also helping to analyze the control flow [7] and information flow of an app [8] in order to track down data leakage. However, despite of all of this information the actual issue still remains, since the user has to grant an app all requested Permissions when s/he wants to use this app, no matter how much s/he distrusts it. For this very reason, a lot of privacy systems are developed with the aim to give users more control over the behavior of their apps by manipulating their access rights.

Nauman et al. [9] introduce with *Apex* an enhanced version of the Android Permission system. The user can select for each Permission whether s/he wants to *allow*, *deny*, or *constrain* it. Since Apex is embedded in the Android platform, it can control the *checkPermission* method, which is responsible for granting or denying access rights. Therefore, any app can be regulated via Apex, but withdrawing a Permission will inevitably lead to a crash of the app. The problem of app crashes due to missing Permissions is addressed by *MockDroid* [10]. Instead of denying a data request, MockDroid sends fake data to the app, if the user does not want the app to know the actual data. Jeon et al. [11] propose an entirely different strategy with their toolset called *Dr. Android & Mr. Hide*. Dr. Android is a rewriter for Dalvik bytecode and Mr. Hide is a service which grants or denies access to certain data. First, an app is analyzed in order to determine which access rights are really required—keep in mind, the Android Permissions are always a set of several access rights. Then, any method call which requires a Permission is redirected to a similar call to the Mr. Hide service. Thereby, Dr. Android & Mr. Hide can provide new fine-grained access rights, applicable for any app, which is converted by Dr. Android. However, the conversion causes that the app can no longer be updated by the app store and the app's signature is changed which entails further complications [12] (see Section III-B). However, to include the monitor component, the code of an app has to be manipulated, which is a violation of copyrights.

Due to our findings, there is no privacy system which supports Permission granting or withdrawing at run-time, guarantees that apps will not crash due to missing Permissions, allows context-sensitive policy rules, supports fake data, provides feedback about the behavior of apps concerning the usage of private data, supports automatic updates via the app store, and supports a remote configuration of the system for inexperienced users. Therefore, the PMP [13] was developed. However, due to its enhanced privacy policy model (see [14]), which is required to provide all of these features, apps have to supply additional information. Thus, the PMP is usable by compatible apps, only—Legacy Apps are not supported out of the box. For this reason, we study how other privacy systems operate with Legacy Apps and derive strategies which are used in the PMP Gatekeeper in order to enable the PMP to deal with any kind of app.

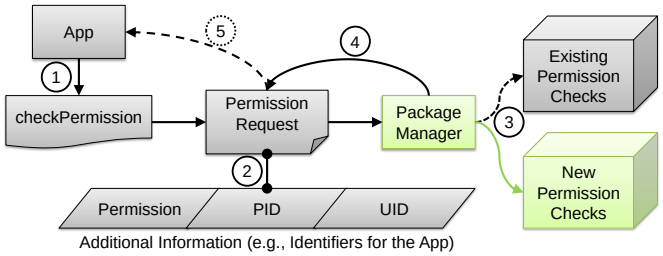

Figure 1. Execution Steps for a Permission Request (New Parts are Green)

III. Strategies for Legacy Apps

Based on the reflection of the related work, there are two different approaches of how to implement a new privacy system. Either the existing privacy system is extended or a monitoring component is implanted into every app. In the following, we describe the characteristics of these two approaches and assess how they can be applied in the PMP.

A. Extension of the Privacy System

By extending the original privacy system, it is possible to introduce improved policy rules. However, the apps still request the unaltered Permissions of the original system. So, a component has to be interposed between the app layer and the privacy system. The interface of this component towards the apps has to be similar to the one of the original system. Internally, the requests have to be replaced by extended ones which can be processed by the new system.

This procedure is simplified depicted in Figure 1. In an unaltered system, an app calls the *checkPermission* method, whenever protected data has to be accessed ①. This method sends out a Permission request to the *PackageManager*. The system adds more information about the app, e. g., its process ID (PID) and its user ID (UID) ②. As every request has to be checked by the *PackageManager*, usually this component is manipulated, in order to alter the original privacy system. Instead of checking the request with the old mechanism, a new component is called ③. After the check, the *PackageManager* sends back either a *PERMISSION_GRANTED* or a *PERMISSION_DENIED* flag ④. If the Permission is denied, a *SecurityException* is raised in the app ⑤.

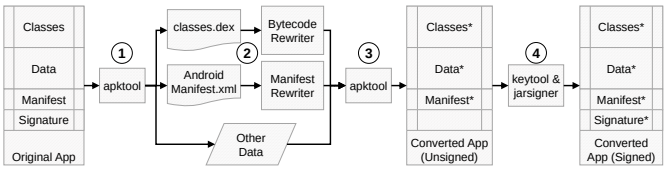

Figure 2. Execution Steps of an App Rewriter

That is the key problem of this implementation strategy. Keep in mind, all Permissions have to be granted at installation time and a subsequent revoke is not possible. So, apps usually do not handle the *SecurityException*. The big advantage of this strategy is, that every app can be regulated out of the box. Therefore, the PMP Gatekeeper makes use of this strategy to deal with apps which are not compatible with the PMP, i. e., Legacy Apps (see Section IV-C).

B. Inline Reference Monitoring

Since every Android app runs in its own sandbox, it is not possible to implement a privacy system as an app monitoring other apps. Therefore, every app has to collaborate with the privacy system (in terms of sending Permission requests to such a monitoring app). As no app innately supports such a monitoring component, any existing app has to be manipulate.

Figure 2 depicts a toolchain with which apps can be rewritten. As an app's source code is not available, it has to work directly on the *apk* (the Android Application Package). Initially, this package has to be extracted and decompiled to restore the original data structure. *apktool* can be used for extracting the apk file ①. After the extraction, any XML parser can be used to manipulate the *AndroidManifest.xml*, e. g., by adding or removing Permission requests. There are bytecode manipulation tools for Dalvik bytecode such as *ASMDEX* ②. After the manipulation, the app's components have to be repacked to an apk file, again using apktool ③. Finally, the app has to be signed using *keytool* and *jarsigner* ④.

This last step leads to two problems: As an app is identified inter alia by its signature, the rewritten app is recognized as a new app. As a consequence, automatic updates are no longer possible, since the app cannot be identified correctly. Moreover, as some rights are bound to the app's signature, such a rewriting process interferes in the rights management of the system. On top of this, the bytecode manipulation is based on assumptions and thus faults can occur (e. g., Permission requests are missed or the bytecode is totally destroyed). So, the user might get lulled into a false sense of security. Additionally, manipulating an app constitutes a copyright violation. Thus it is not a recommendable strategy for the end-user. However, we can support app developers with a toolchain in order to support them to convert their own Legacy Apps to PMP-Compatible Apps.

IV. The PMP Gatekeeper

The **P**rivacy **M**anagement **P**latform (*PMP*) is a context-aware and crash-proof privacy management system in which a user can make fine-grained adjustments to an app's Permissions at run-time and obfuscate or even randomize his or her private data on demand. The user is always informed about what impact his or her settings have upon the affected app's scope of service. In order to realize these features, the PMP bases on an enhanced app model (see [14]). In this model an app consists of several *Service Features* each encapsulating a certain service. Permissions (mapped to *Resources*) are not directly assigned to an app, but to its Service Features. Therefore, the user can deny some Permissions and still use the app, since each Service Feature can be enabled or disabled individually. For further information about the components of the PMP and its functionality, please refer to the literature [5].

In order to facilitate all of these features, an app has to provide additional meta data, e. g., for the specification of its Service Features. The key aspects of *PMP-Compatible Apps* are discussed in the following. Thereafter, a description is given of how the PMP Gatekeeper can differentiate between PMP-Compatible Apps, Legacy Apps, and System Apps and how it revokes Permission for Legacy Apps.

A. PMP-Compatible Apps

PMP-Compatible Apps have three characteristics:

- (a) PMP-Compatible Apps have to register at the PMP. Therefore, an entry in the global *AndroidManifest.xml* is required in order to evoke the PMP's registration call.
- (b) Within the app's source code, Permission requests are sent directly to the PMP and the protected data is sent back by the responsible Resource via *IPC*.
- (c) The Service Features have to be defined and the required Resources have to be specified. This data is stored in the *Application Information Set* (*ais.xml*).

In the *ais.xml* the app itself is described by its name and an optional description for identification reasons. In a second block the Service Features are defined. For each feature a unique identifier and a feature name has to be given. Also, a description of the feature is required, so that the user gets an idea of how much service quality s/he loses by deactivating a certain feature. Lastly, the required Resources and Privacy Settings (i. e., its Permissions) for that Service Feature have to be specified. The values of the Privacy Settings do not have to be boolean values, only. Any given data type can be used to describe restrictions as fine-grained as possible (Requirement (1)).

Each of these three characteristics can be used to differentiate between PMP-Compatible Apps and Legacy Apps.

B. Classification of Apps

The PMP Gatekeeper is a totally new PMP component which realizes an access control mechanism for Legacy Apps, i. e., all apps which cannot be controlled by the PMP hitherto. For these Legacy Apps, the PMP Gatekeeper grants or denies data access based on fine-granular user-defined policy rules. However, since System Apps—i. e., any app which is part of the mobile platform—are by definition also Legacy Apps, too restrictive policy rules could have serious consequences for the system's stability and performance. Accordingly, it is a key

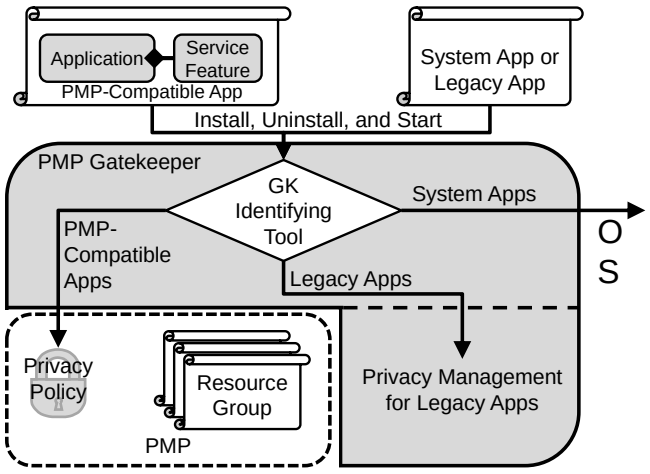

Figure 3. Logical Inclusion of the PMP Gatekeeper into the System

task of the PMP Gatekeeper to classify PMP-Compatible Apps, Legacy Apps, and also System Apps unfailingly.

Since the apk file does not include any source code, the identification of PMP-Compatible Apps works via the characteristics (a) and (c). Initially, the PMP Gatekeeper requests an app's package information. This information contains all data from the manifest and can be accessed with the *PackageManager.getPackageInfo* method. If the manifest contains the PMP registration activity, then the PMP Gatekeeper removes all Permission tags, since they are not needed for PMP-Compatible Apps. To double-check that the app is a PMP-Compatible App, the PMP Gatekeeper searches for the app's ais file. The platform's *AssetManager* provides methods to access the app's assets and the PMP Gatekeeper validates them. If one of the two checks fails, the app cannot be a PMP-Compatible App.

In order to decide whether the vetted app is a Legacy App or a System App, the PMP Gatekeeper has to check the package information again. System Apps are stored in a special partition and are tagged with a *FLAG_SYSTEM* flag. Essentially, a check for this flag is sufficient to identify System Apps. However, since the platform vendor sometimes assemble third party apps into the OS (e. g., the *Facebook* app), the PMP Gatekeeper enables users to specify certain System Apps which should be monitored and regulated as well. The identification mechanism is resource-efficient and thus, neither the stability nor the performance of the system is affected (Requirement (3)). Figure 3 shows how the PMP Gatekeeper is integrated into the system.

C. Revoke of Permissions

The PMP Gatekeeper is not only able to identify third party Legacy Apps, but also to intercept access attempts to private data for which the user has revoked the Permission. Simply removing the Permission tag from the manifest does not have the desired outcome since these entries are only analyzed at installation time. Afterwards, the Permissions of an app are stored within the OS. It is possible to manipulate the manifest followed up by uninstalling and installing the modified app, but this is ineligible when Permission changes at run-time are intended. The commonly suggested solution to manipulate the Permissions entries within the OS is also inadvisable, since these entries are stored in secured parts of the main memory which are only written to a backup file (*packages.xml*) when the system is shut down. However, this file has neither a coherent data model—i. e., depending on the respective app the Permissions are stored in different ways—nor does it contains the Permission information for all apps. Some information are outsourced to completely different files for no obvious reasons. The manipulation of the backup file at run-time would have no effect on the app's Permissions: On the one hand changes are noticeable after a reboot, only, and on the other hand all changes are overwritten when the system shuts down.

Thus, the PMP Gatekeeper requires a different strategy. We copy the basic idea of a policy which is held in main memory all the time and is only made persistent at a certain point in time. However, as the PMP Gatekeeper is the owner of this policy, it is able to make changes at run-time to both the stored file as well as the policy held in memory. At run-time the PMP Gatekeeper determines all Permissions an app specifies in the app's manifest and grants the app each of them. Upon request, the user can individually withdraw (or grant) Permissions at run-time. When an app sends out a Permissions request the PMP Gatekeeper simply has to intercept the request at the *PackageManager* and refer it to the PMP Gatekeeper's policy (see Figure 1). Depending on the user's settings, the access is either granted or denied. However, in order to ensure a crash free execution of the app, no *SecurityException* is raised, but fake data is sent to the app. More implementation insights are given in Section V.

The modus operandi of the PMP Gatekeeper is as follows: For Legacy Apps, the user gets an overview of all installed third party apps and s/he can look up all Permissions any of them requires. Since the shown Permissions are derived from the manifest directly, this list is complete and not an aggregated summery as it is used in the original Android installation dialog. The user can add or revoke any Permission in this list. By default, all Permissions are granted to guarantee an unrestricted functionality for any app.

As a consequence, any kind of app can now be monitored and regulated by the PMP. PMP-Compatible Apps are supported by the PMP out of the box and Legacy Apps can be dealt with by the PMP Gatekeeper. Even the Permissions of System Apps can be restricted (Requirement (2)).

V. Implementation Insights

After the overview of the PMP Gatekeeper's functionality, this section provides details on two implementation aspects: Section V-A describes how the PMP Gatekeeper is hooked into the PMP while Section V-B explains the required alternations towards the mobile platform.

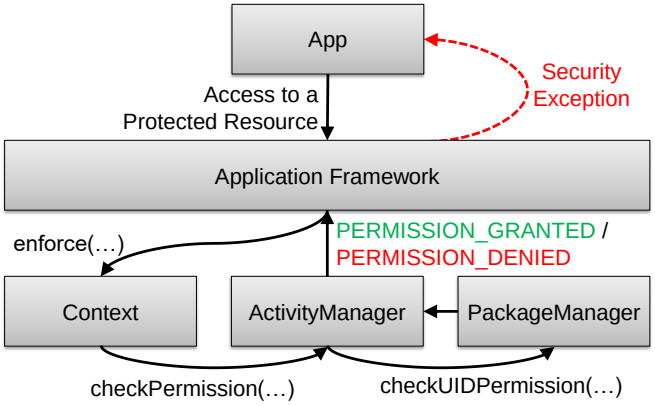

Figure 4. Android Call Chain for Permissions Requests

Listing 1. Code Enhancements towards the *checkPermission* Method

```
checkPermission(String permission, int pid,
    int uid) {
  if (permission == null) {
    return PackageManager.PERMISSION_DENIED;
  }
  if (checkComponentPermission(permission,pid,
      uid,-1) == PackageManager.
      PERMISSION_GRANTED) {
    String[] packageNames = PackageManager.
        getPackagesForUid(uid);
    return checkGatekeeper(packageNames,
        permission);
  } else {
    return PackageManager.PERMISSION_DENIED;
  }
}
```

A. Interfaces Towards the PMP

The *Android Application Framework* responds to various system events (e. g., an incoming SMS) by sending so-called *Broadcasts*. These system notifications can be intercepted and processed by any app which defines a corresponding *BroadcastReceiver* in its manifest. However not only hardware events result in a Broadcast, but also software events, e. g., actions of the *PackageManager* such as installing or uninstalling an app. For this reason, the PMP implements an *InstallReceiver* and an *UninstallReceiver* responding to the installation and the removal of apps via the *PackageManager* by passing the names of the apps to the PMP.

Thus, the interface to the new PMP Gatekeeper component can be hooked into the *InstallReceiver*. Whereas hitherto the PMP assumed that any app is a PMP-Compatible App and hence tried to install it via the PMP installation dialog, the PMP Gatekeeper initially identifies the app's type: When an app is installed the *PackageManager* initiates the *InstallReceiver* via a *PACKAGE_ADDED Broadcast Intent*. The *InstallReceiver* forwards the information about the app to the PMP Gatekeeper. The PMP Gatekeeper checks whether the app is a PMP-Compatible App or a Legacy App (see Section IV-B). For PMP-Compatible Apps the PMP Gatekeeper invokes the PMP's installation and registration mechanisms. Legacy Apps are added to the PMP Gatekeeper's Legacy Apps list. For each list entry, the PMP Gatekeeper determines all Permissions and adds them to its whitelist as initially an app's access rights are not restricted. This can be revised by the user, subsequently.

The uninstall process triggers a *PACKAGE_REMOVED Broadcast Intent* after removing the app. Therefore, the PMP Gatekeeper has the app's name, but can no longer analyze the app and identify whether the app is a PMP-Compatible App or a Legacy App since the apk is gone already. However, this is not necessary anyway. The PMP Gatekeeper only has to check whether the app still has an entry in the Legacy Apps list and remove this entry to keep the list lean.

It is not necessary to identify System Apps in this working step as System Apps cannot be uninstalled. Even if the user removes a System App manually (e. g., by acquiring root rights and then deleting the apk), this would not pose a problem for the PMP Gatekeeper, since additional entries increases the list's size only marginally. Furthermore, as System Apps cannot be installed by the user (they are directly shipped and installed with the platform) only a negligible number of them appears in the PMP Gatekeeper's list anyhow.

B. Extension of the Mobile Platform

As described in Section IV-C the only practicable approach to revoke a Permission is to enhance the *Android Application Framework* as any data request is processed within this framework. Figure 4 depicts a simplified sequence of the method calls which are required for a Permission check. Basically each of the three methods (*enforce*, *checkPermission*, and *checkUIDPermission*) is qualified to be a host for the PMP Gatekeeper. Our implementation strategy reads as follows: *As early as possible, as late as necessary.*

The sooner the PMP Gatekeeper sorts out apps, the less apps have to be checked in the subsequent steps which saves processing time. However, early steps in the process chain (e. g. *enforce*) have very little information about the requesting app. These information arise in the *ActivityManager*—namely the *UID* required to identify an app. Therefore, the PMP Gatekeeper is hooked into the *checkPermission* method. Listing 1 shows how this is done: First, the PMP Gatekeeper checks whether the unaltered privacy system would grant the Permission. Only if that is the case, the PMP Gatekeeper gets active. Instead of passing the result of the *PackageManager* to the framework, it starts its own validation: After the identification of the app type, PMP-Compatible Apps are forwarded to the PMP without further checks. Likewise, System Apps are passed through to the OS without further ado. For Legacy Apps the PMP Gatekeeper checks in the whitelist, whether the Permission is granted.

VI. Assessment

So, the PMP Gatekeeper fulfills the requirements towards a holistic privacy system: (1) The extended PMP enables its users to grant or deny any given subset of the Permissions an app requests at any time. Therewith, a fine-grained configuration of

the privacy system is possible. (2) The extended PMP monitors any kind of app. The PMP Gatekeeper ensures that each type of app is dealt with by the appropriate PMP component. Even System Apps can be monitored. (3) The classification mechanism is unambiguously and resource-efficient. So, neither the system's stability nor its performance is sustainably affected by the PMP Gatekeeper.

With the PMP Gatekeeper, the user has total control over any private data. S/he is free to decide which information should be process by a certain app. Since these Permissions can be fine-grained adjusted arbitrarily at run-time, the user sees immediately what effect a specific modification has on the scope of service of the affected app. So s/he can find the optimal balance between security and functionality.

VII. Conclusion

Today, there is an app for virtually any use case. As a consequence, a lot of sensitive data is stored on smart phones. Startled by an increasing number of information thefts and data misuse incidents, smart phone users are becoming more and more aware of the need for comprehensive protection measures for their sensitive data in order to maintain their privacy. Although the mobile platform vendors are aware of their responsibility to protect their users' data, their solution approaches seem to be halfhearted. This is hardly surprising since the mobile platform vendors cannot spoil things with the app developers whose business model is often focused on collection as much as possible data in order to use it for personalized ads. Therefore, the better approaches—in terms of more secure approaches—originate from third parties. Their common problem is, that every app is aligned with the privacy system of the mobile platform and it does not collaborate with any other approach out of the box.

For this reason, we discuss different approaches of how to deal with Legacy Apps—namely manipulating the original privacy system or manipulating the apps. Since none of the existing approaches seems to provide a comprehensive privacy mechanism for any kind of app, we introduce the PMP Gatekeeper as an extension for the PMP. With this new component, the PMP is able to provide its context-aware, fine-grained, and crash-proof privacy management functionality not only for PMP-Compatible Apps but also for Legacy Apps without impairing the OS. We give implementation insights of the main components of the PMP Gatekeeper and show that the PMP Gatekeeper meets all requirements towards a holistic privacy system. Moreover, we also indicate how the findings of the related work study can be used to create a toolchain to convert Legacy Apps into PMP-Compatible Apps in order to gain even more control over their data usage. Although the presented prototype is implemented for Android, the underlying concept can be applied to other application platforms such as the *Facebook Platform* or *Chrome OS*.

Acknowledgements

The PMP results from a close collaboration with Google Munich office. Hence, we would like to thank Google for their support. We also thank our student Diana Salsa for her assistance with the implementation of the PMP Gatekeeper.

References

[1] C. Stach and A. Brodt, "—vHike—A Dynamic Ride-sharing Service for Smartphones," in *MDM'11*, 2011.

[2] C. Stach, "Saving time, money and the environment - vHike a dynamic ride-sharing service for mobile devices," in *PERCOM Workshops'11*, 2011.

[3] C. Stach and L. F. Schlindwein, "Candy Castle - A Prototype for Pervasive Health Games," in *PERCOM Workshops'12*, 2012.

[4] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android Permissions: User Attention, Comprehension, and Behavior," in *SOUPS'12*, 2012.

[5] C. Stach and B. Mitschang, "Design and Implementation of the Privacy Management Platform," in *MDM'14*, 2014.

[6] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android Permissions: A Perspective Combining Risks and Benefits," in *SACMAT'12*, 2012.

[7] P. P. Chan, L. C. Hui, and S. M. Yiu, "DroidChecker: Analyzing Android Applications for Capability Leak," in *WISEC'12*, 2012.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, 2014.

[9] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *ASIACCS'10*, 2010.

[10] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading Privacy for Application Functionality on Smartphones," in *HotMobile'11*, 2011.

[11] 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 *SPSM'12*, 2012.

[12] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[13] C. Stach, "How to Assure Privacy on Android Phones and Devices?" In *MDM'13*, 2013.

[14] C. Stach and B. Mitschang, "Privacy Management for Mobile Platforms—A Review of Concepts and Approaches," in *MDM'13*, 2013.