Technical Report

Nr. TUD-CS-2012-0226
November 1st, 2012

CASED

myTunes:
Semantically Linked and User-Centric Fine-Grained Privacy Control on Android

Authors
Sven Bugiel, Stephan Heuser, Ahmad-Reza Sadeghi

myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android

Sven Bugiel[*], Stephan Heuser[†],
Ahmad-Reza Sadeghi[*,†,‡]

[*]Technische Universität Darmstadt
Darmstadt, Germany

[†]Fraunhofer SIT
Darmstadt, Germany

[‡]Intel Collaborative Research Institute for Secure Computing
at Technische Universität Darmstadt, Germany

ABSTRACT

Smartphone operating systems provide designated databases and services for user's private information (e.g., contacts data and SMS or location) that can be conveniently accessed by 3rd party apps through clearly defined APIs. The popular Android OS deploys a permission framework and a reference monitor to protect the interfaces of these security and privacy sensitive components. However, Android's default permissions are too coarse-grained and inflexible to provide sufficient protection of users' privacy as recent privacy violation incidents show. Recently several privacy-protection solutions on Android have been proposed. However, all of these solutions regulate access to sensitive information on a binary all-or-nothing basis.

In this paper, we present the design and implementation of a user-centric fine-grained privacy control on Android's system ContentProviders and Services. Our solution tackles the challenge of providing a *per-data* and per-app access control to private information. This allows the user to intuitively adjust his privacy settings (by means of a Privacy Control App) with respect to both his need to share and protect private information. Moreover, a particular contribution of our work is establishing semantic links between the access control rules in different system components to prevent inferring of sensitive information from deputy databases or services. We successfully evaluate the effectiveness and efficiency of our solution based on popular social networking apps.

1. INTRODUCTION

Smart Devices such as Smartphones and Tablets have become an integral part of our daily life and communication. The most important factor for the success of Smart Devices is the extremely convenient way in which even technical non-savvy end-users are able to fully utilize the power and features of these devices. The foundation for this success is the ecosystem that has evolved around Smart Devices: end-users can choose from a plethora of apps provided on App Markets/Stores, interconnect their devices for synchronization and service sharing, or connect to a diversity of Cloud-based services.

As a drawback of this evolution, also malware (or spyware) authors have discovered Smart Devices as appealing targets for attacks. Researchers have shown that end-user mobile devices are vulnerable to a number of different attacks – ranging from malicious over-privileged apps [22] and nosy advertisement libraries [18, 14] extracting sensitive private information, like contacts, SMS/MMS or the current location, to botnet-like behavior [31] utilizing root exploits [3, 4]. In particular the Android Operating System (referred to as AOSP) has been the most favorite target of industrial and academic research due to its open source character that allows modifications at all layers of the software stack. This research has thoroughly scrutinized Android's security mechanisms and disclosed a number of vulnerabilities and shortcomings.

User-centric filtering of information. In this paper, we focus on enabling a more fine-grained filtering of private information that can be retrieved from information stores (*ContentProviders*) and system Services on Android.

Android's design facilitates clearly designed interfaces to store and retrieve sensitive information. For instance, retrieving the physical location of the mobile device, polling the motion sensor data, or getting the IMEI number are performed by means of remote procedure calls to the corresponding system Service like LocationManager, SensorManager, or TelephonyManager, respectively.

A ContentProvider, on the other hand, is an abstraction of an information storage, such as an SQLite Database, which provides common interfaces for inserting, updating, deleting, and querying data to and from the underlying information storage. ContentProviders are the primary means for apps to organize and share data on Android. Popular providers on a stock Android installation storing sensitive private information are, for example, the ContactsProvider or the MMSProvider (which also handles SMS).

Since the system Services and ContentProviders handle security and privacy sensitive data, read and write access to them is regulated based on permissions (e.g., `READ_CONTACTS` and `WRITE_CONTACTS` or `ACCESS_FINE_LOCATION`). As usual on Android these permissions must be requested by 3rd party applications. The user has to confirm them during app installation and cannot selectively disable or revoke them on stock Android. These permissions regulate the overall access to ContentProviders and Services in a binary "all-or-nothing" policy decision.

This rather limited flexibility and coarse-grained access control has been in the focus of recent privacy incidents. For instance, it was shown that advertisement libraries frequently included in 3rd party apps on Android leverage the

permissions of their host app to gather information about the user [18]. They also relate to controversial discussions about the popular WhatsApp [7, 6] Path [5], and Facebook [2] apps, which have been alleged to clearly overstep the necessary boundaries of their access to user's private data, violating the user's privacy. These incidents motivate the need for configurable, user-centric *least-privilege* and fine-grained access control for apps to privacy sensitive data provided by system ContentProviders and Services.

Fine-grained acccess to private information. Related work on Android's security mechanisms has acknowledged these privacy problems and presents solutions, e.g., to gracefully revoke apps' permissions to access a ContentProvider or prevent this data from being exfiltrated [32, 20, 25, 12, 10]; to establish per-data access control based on delegable policies for DRM [26]; or to inline a reference monitor in 3rd party application code for privacy control [30, 9]. In fact, some of those ideas have been adopted in popular community mods like CyanogenMod[1] or commercial products like Whisper Systems [8] and 3LM [1]. However, while these solutions provide revocable access to private data, the access control still remains binary—either an app can access and use all data provided by a ContentProvider/Service or no (benign) data at all. Although this essentially protects the privacy of the user, it also seriously impedes the user-desired functionality of apps. For instance, if WhatsApp could no longer access the user's contacts' names and phone numbers, it would be completely incapacitated and would not provide any add-value to the user. Thus, a practical solution requires rather a fine-grained *per-data* than a *per-interface* access control. Controlling 3rd party apps access to sensitive information on a per-data basis is essential for empowering the user to adjust his privacy settings with respect to both his desire to protect his data and simultaneously his need to share his data. A binary all-or-nothing control, as currently provided by default Android or related work, cannot address both needs. Moreover, in case of system ContentProviders, access to information is not necessarily restricted to a particular designated ContentProvider, but could also be retrieved indirectly from another provider. For example, retrieving the call log of the user, mining the phone numbers in the SMS database or collecting the email addresses from the EmailProvider discloses information about the user's contacts (e.g., their numbers, names and email addresses). Such *semantic links* between different providers have, to the best of our knowledge, not been considered previously.

Our contribution. In this work, we extend the default system ContentProviders and Services on Android with a per-data access control and empower the user of the device to adjust at runtime to which specific data a 3rd party app should have access. This selection can be based on, e.g., data types such as mimetypes. We extend Android's system Services with a *per-data* access control in comparison to the default permissions and present for ContentProviders a new design pattern for access control, which is based on SQL Views within the databases. A particular benefit of this approach is, that the access control dynamically autoconfigures itself to new data types at runtime and thus always provides the most recent and fine-grained access control configuration. The access control can be configured by the user by means of a dedicated *Privacy Control App*, which is responsible for managing and deploying the access control rules and presenting them in an intuitive way to the user. Self-contained access control within the ContentProviders and Services ensures that only this trusted front-end app (i.e. the user) can configure the access control rules.

Our second contribution concerns semantic links among the stock system ContentProviders and Services, i.e., where data from one provider can be indirectly retrieved through a deputy provider or service. We implement in our solution a mechanism for implicit semantically-linked access control that reflects access control decisions by the user for one provider in the access control rules of other, linked providers. For instance, if the user denies an app access to his "work" contacts, access to SMS and call log entries related to this contacts group is denied automatically as well.

Lastly, we successfully verified the effectiveness and practicability of our solution with popular social networking apps, such as *Facebook* and *WhatsApp*. Further, our tests confirm the above stated problems of related work.

Although we present and evaluate in this paper a standalone implementation of our *per-data* access control framework, it can be easily integrated or combined with related work to achieve more dynamic and flexible policy enforcement (e.g., considering the system state and phone context). To this end, we plan to release the code of our implementation and hope that it will contribute in the future to the security and privacy-protecting features of community ROMs such as CyanogenMod or even of AOSP.

2. ANDROID

Android is an open source software stack for mobile devices, such as smartphones or tablets. It comprises of a modified Linux kernel, the Android middleware, and an application layer. While the Linux kernel provides basic facilities such as memory management, process scheduling, device drivers, and a file system, the middleware layer consists of native system libraries, the Android runtime environment including the Dalvik virtual machine and an application framework. The Android application framework consists of system applications implemented in C/C++ or Java, such as system ContentProviders and Services. These provide the basic functionalities and the essential services of the platform, for instance, the Contacts app, the Clipboard, the SystemSettings, the AudioManager, the WifiManager or the LocationManager.

In the following we selectively provide the necessary background information on Android for the understanding of our problem description and solution. In particular, we elaborate in more detail on the available mechanisms for organizing the user's data and how this data is accessed and shared between different applications. We further explain the related specifics of Android's security mechanisms to provide basic access control to this data.

2.1 Security Mechanisms

Android implements a number of security mechanisms, most prominently application sandboxing and a permission framework that enforces access control on inter-app communication and on the access to core functionalities. In the following, we provide a brief summary of these mechanisms and refer to [15] for a more detailed discussion.

Android applications are signed using a private key bound

[1]Cf. Section "Permission Management" at `http://www.cyanogenmod.com/about`

to the developer by means of a (usually) self-signed certificate. During installation, apps are assigned a unique Linux user identifier (UID) which is used to provide an app-private storage directory and to sandbox the app process at runtime. Multiple apps signed by the same key can share a UID.

At runtime, Android enforces access control on inter-app communication. The access control mechanism is based on *Permissions* [16] which an application must request from the user during installation. The user has to grant *all* requested permissions in order to allow the app installation or alternatively deny the installation. Permissions cannot be selectively granted or denied. Since technically permissions are bound to UIDs, apps sharing a UID also share their granted permissions. Android already contains a set of predefined permissions for access to system resources [16], but applications can also define new, custom permissions to restrict access to their own interfaces. A reference monitor in the Android middleware checks if an application holds the necessary permissions to perform a certain protected action, for instance, to bind to a protected service or to start a protected other application. Permissions controlling access to kernel-level ressources, such as the ability to open sockets, are mapped to Linux group identifiers (GIDs) assigned to an app during installation and checked at runtime by reference monitors in the modified Linux kernel.

2.2 ContentProviders

ContentProviders are the primary means to organize and share data between applications in a secure way [17].

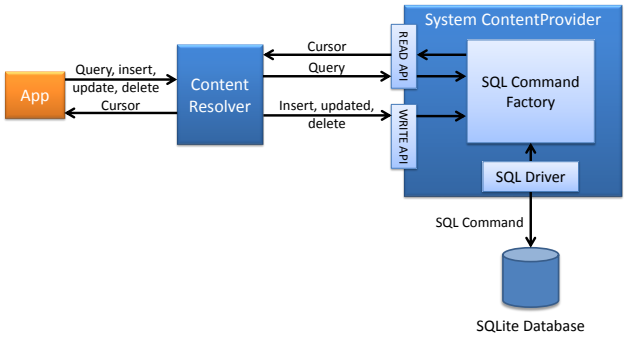


Figure 1: ContentProvider workflow

Structure and workflow. ContentProviders encapsulate a structured set of data, for instance stored in back-end storage, and provide clearly defined interfaces to other applications to access this data (cf. Figure 1). The exact back-end storage can vary, however, the most common and for system ContentProviders default option is an SQLite database back-end. The provider's interface resembles an SQL-like interface, i.e., it provides `query`, `insert`, `update`, and `delete` functions to manage the data. Usually, each of those functions requires a fixed set of parameters (e.g., selection criteria or sort order for queried data), which are used by a provider-internal command factory to assemble the actual SQL command that is issued to the back-end database.

The default mechanism to connect to a ContentProvider is the so called *ContentResolver* class. Each ContentProvider can be addressed by an URI-like, system-unique address, the *Content URI*, e.g., `content://contacts` for the system ContactsProvider. This URI can also refer explicitly to tables or entries in the providers database. When an app issues a command to a ContentProvider, the ContentResolver resolves the URI of this command to the corresponding provider and relays the command and its reply between provider and app. Data returned to apps upon queries is presented in form of so called *Cursor* data structures, which represent the results in a table according to the function parameters (e.g., selection or ordering).

ContentProvider Security. To access the system ContentProviders, for example the ContactsProvider or the SMSProvider, permissions are required by default. For ContentProviders, Android generally distinguishes between permissions for read access (e.g., `READ_CONTACTS`) and write access (e.g., `WRITE_CONTACTS`).

In addition, Android provides the *URI Permission* mechanism for ContentProviders. It allows a ContentProvider or its client app to dynamically delegate the permission to access a specific data (identified by the URI) to another app at runtime. For instance, this mechanism is useful when an email app wants to provide access to an email's attachment (e.g., a photo) to an auxiliary app (e.g., picture viewer app), without requiring the auxiliary app to hold the permission to access the entire ContentProvider data set.

2.3 Services

Services are usually background processes, which provide remotely (i.e., inter-process) callable function that are specific to each Service.

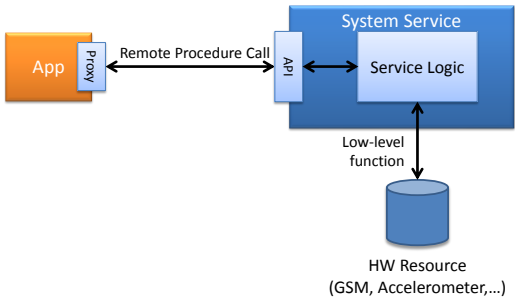


Figure 2: System Service workflow

Structure and workflow. System services in particular encapsulate specific hardware resources or devices. Prominent system Services are, for instance, the LocationManager that provides functions to retrieve the device's physical location from the GPS module, the SensorManager that provides functions to poll the on-board motion sensors, or the TelephonyManager that communicates with the SIM Card and GSM module.

The API that a Service exposes must be defined by means of the Android Interface Definition Language (AIDL). Callers to this API retrieve from the Service a Binder IPC *proxy* object, which provides the API and is responsible for wrapping and forwarding calls and responses of API calls between caller and Service.

Service Security. Similar to system ContentProviders, permissions are required to access system Services, for example, `ACCESS_FINE_LOCATION` to retrieve the GPS coordinates of the device or `READ_PHONE_STATE` to get the IMEI number. While the permission can be assigned to a Service component as a whole (the caller must hold the permission independently of the called function), it is more common to

assign different security levels to the exposed functions and thus perform the permission check (with possibly different permissions) at the granularity of functions. For instance, the TelephonyManager requires either the `READ_PHONE_STATE`, `ACCESS_COARSE/FINE_LOCATION`, or no permission from the caller, depending on the called function.

3. PROBLEM DESCRIPTION

In this section we describe the conceptual privacy problem of Android that we address in our solution (cf. Section 3.1) and the technical challenges to be tackled (cf. Section 3.2).

3.1 Privacy Problems

Smart devices store and process a high number of private information. On Android, two crucial problems must be addressed to provide an efficient and effective protection of the user's privacy.

Coarse-grained and irrevocable default permissions. It has been shown, that Android's default permission model defines too coarse-grained permissions to efficiently protect user's privacy. For instance, apps holding the `READ_CONTACTS` permission have access to *all* contact data including the data irrelevant for the apps' correct functioning. Moreover, once granted by the user, permissions are irrevocable until the app is uninstalled. In light of recent news regarding users' privacy violations [7, 6], it is thus highly desirable to provide users with a fine-grained and adjustable access control over their private information.

Least-privilege. The problem of coarse-grained and irrevocable permissions has been identified previously and, as we explain in detail in related work (cf. Section 7), different approaches aim at addressing this issue [32, 30, 9, 21, 10, 11, 26, 13, 20, 25, 12, 27, 23]. These approaches provide additional access control to information sources such as (system) ContentProviders or (system) Services, which can be adjusted at runtime, or provide a so-called data shadowing mechanism that returns fake data on access to information stores. However, these mechanisms do not provide a *least-privilege* access to private data, but rather regulate access to entire information stores on an all-or-nothing basis and thus seriously impede an app's user-desired functionality. For instance, preventing a WhatsApp-like messenger from accessing or exfiltrating any contact data surely protects the user's privacy, but also blocks the app from building a list of contacts with whom the user can exchange instant messages and hence prevents the app from providing the user-desired messaging service.

Semantic links between information stores. Moreover, related mechanisms as described above ignore the fact that lots of information can be indirectly retrieved from other information stores, thus rendering their provided access control ineffective. Extending the previous example, the WhatsApp-like app could retrieve required phone numbers from the CallLog or SMS database (if it holds the corresponding permission) instead of directly from the ContactsProvider. CallLog and SMS provider effectively act as deputies in this scenario.

Thus, in practice it is desirable to have an *per-data* access control to private information, which empowers the user to adjust his privacy settings with respect to both his need to protect his data and simultaneously his need to share his data. Unfortunately, default Android and related work currently do not provide such a mechanism. Additionally, this mechanism should exploit the static relations between (system) ContentProviders to semantically link access control between different providers (e.g., an app that can read business contacts, should simultaneously not be able to read those contact's SMS messages).

3.2 Technical Challenges

To provide an efficient and effective solution to the above described privacy problems, certain technical challenges must be solved.

Fine-grained access control. The access control mechanism has to be fine-grained enough to support rules on a per-data basis. Android ContentProviders structure data in SQLite tables. The granularity of the access control should be per row, column, or cell of the table. For instance, our WhatsApp-like example app should only retrieve the columns containing the phone number and name of contacts. If additionally the user requests that this app should only access contacts in the group *Friends*, all data rows not belonging to Friends contacts must be inaccessible to this app or removed from query responses to this app.

User-centric configuration. A particular challenge for a user-centric configuration of the access control is how the access control rules can be presented to the user for intuitive configuration.

Low performance impact. Despite access control on a per-data basis and potentially big databases, the performance impact should be minimal. Thus, the access control mechanism should be ideally placed in the database to leverage its native speed.

4. FINE-GRAINED PRIVACY CONTROL

In this section, we present the design of our solution. We give a high-level overview of our architecture in Section 4.1, explain the access control mechanisms in our architecture in Section 4.2, discuss alternative policy deployment 4.3, and present the semantic links between different access control rules in Section 4.4.

4.1 Overview

The goal of our architecture is to facilitate a *per-data* instead of a per-interface access control to privacy sensitive information provided by system ContentProviders (e.g., contacts, SMS/MMS, call log) and Services (e.g., location, IMEI, sensor data). Providing a per-data access control allows the user to resolve the conflict between his need to share private data and his need to protect his privacy according to his own privacy requirements by configuring a least-necessary access to his data. For instance, consider the case of a WhatsApp-like messenger, which clearly has to upload the user's contacts' phone numbers to a server to retrieve the list of contacts numbers which are also using this messenger service. Additionally, the app should be able to access the contacts' names to display the user's messenger contacts by name and not only by number. However, there is no necessity for this app to infer the user's relation to his contacts (e.g., friends, family, work colleagues) or to retrieve further sensitive information such as the contacts' postal addresses or email addresses.

An overview of our architecture to establish such fine-grained access control on user's private information is depicted in Figure 3. The user can configure the least-necessary access of apps to his data by means of a *Privacy Control App*,

which then deploys the corresponding access control rules to a new enforcement system in the system ContentProviders and Services. These enforcement points are responsible for controlling 3rd party apps' access to privacy sensitive information provided by information stores, such as databases, or hardware resources, such as GPS or accelerometer, on a per-data basis.

Conceptually, when comparing our high-level design to established access control frameworks such as SELinux [24], our Privacy Control App performs the role of a Security Server (i.e., policy decision point) and our filtering points the role of Object Managers (i.e., object-specific policy enforcement points). However, in contrast to those systems, our policies are static enough to be deployed directly at the enforcement points after configuration/initialization in the Privacy Control App, thus benefiting the overall performance of our system.

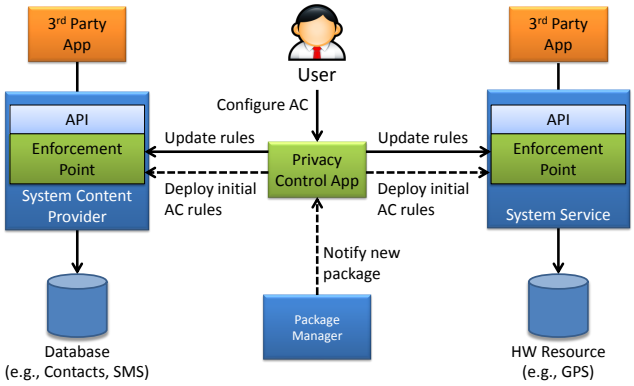


Figure 3: Overview architecture

4.2 Access Control

In this section, we explain how the access control rules in our system are formatted and initialized, as well as configured by the user and eventually enforced.

4.2.1 Rules Format

Our policy rules express a subject-object access matrix of the form

$$\mathbb{S} \times \mathbb{O} \mapsto 0/1$$

where $\mathbb{S}$ is the set of all subjects in the system (i.e., 3rd party apps), $\mathbb{O}$ is the set of all objects in the system (i.e., data), and the mapping is the access control decision, i.e., either deny (0) or allow (1). An access control rule r in this matrix is of the form $r_{s,o} := s \times o \mapsto 0/1$, which describes that a specific subject s has (1)/has not (0) access to a specific object o. We build one access matrix for each enforcement point, i.e., content provider or service. For instance, in case of the ContactsProvider, $\mathbb{S}$ is the set of all 3rd party apps holding the default permission to access the provider (`READ_CONTACTS` / `WRITE_CONTACTS`) and $\mathbb{O}$ is the available data in the content provider differentiated by, e.g., mimetype, group-membership, or account. Access control is then of the form

$$\mathcal{AC}(s, o) \mapsto 0/1$$

where $s \in \mathbb{S}$, $o \in \mathbb{O}$, and $\mathcal{AC}$ is a look-up function which returns the access control decision for $r_{s,o}$ from the set of access control rules or a default value $\perp = 0/1$ otherwise. In our system, we set $\perp = 1$, meaning we allow access by default. This is motivated purely by usability reasons to not affect 3rd party apps in case of misconfiguration, however, we note that $\perp$ could be easily set to 0 if privacy concerns outweigh usability concerns.

4.2.2 Initial Access Control Rules

Since the set of 3rd party apps, which will be installed later, cannot be known *a priori*, our architecture requires a mechanism to initialize the access control rules for newly installed 3rd party apps at runtime.

In our architecture, we leverage the default Android PackageManager to detect newly installed applications (cf. Figure 3). The PackageManager is responsible for (un-)installing apps and broadcasts after a successful (un-)installation a notification to inform other packages about this event. Our Privacy Control App registers itself as a receiver for this broadcast message. Upon installation of a new app, the Privacy Control App inquires from the PackageManager the set of permissions which the new app has been granted and determines from these permissions which system ContentProviders and Services this app has access to. Providers and services that are not protected by a permission are assumed implicitly as being accessible.

To each provider and service that is accessible by the new app, the Privacy Control App deploys new set R of access control rules $R_{s,O} = s \times O \mapsto \perp$ specific to the app, i.e., the subject s of each rule is the UID of the new app and the set of objects O of each rule are the distinct data types managed by the provider or service (e.g., contacts information or location data). Initially, each rule has the default value $\perp$ as access control decision, until adjusted by the user via the Privacy Control App. Performing this deployment directly upon receiving the broadcasted notification, ensures that the access control rules are deployed *before* the app contacts the provider/service the first time and thus that all access is correctly controlled.

4.2.3 Configuration

After deployment of the initial access control rules for an app, the user can change these rules according to his privacy needs by adjusting the access control decision value for each rule (cf. Figure 3). The Privacy Control App takes here on the role of a central configuration tool by displaying a graphical representation of the access matrices in the system to the user, where columns represent the objects, rows the subjects, and cells contain the decision value of each rule identified by its subject-object combination. The user can then select and change the values of each cell. Updated rules are propagated by the Privacy Control App to the corresponding enforcement points.

Similarly, if an app is uninstalled, the Privacy Control App removes all access control rules for this app from the access matrices of the content providers and services.

4.2.4 Enforcement

Our design decision for the access control enforcement is, that enforcement is *gracefully*, i.e., it does not simply revoke the permission to access certain data at runtime and throw an unexpected security exception back to the calling app, but instead applies transparent data filtering or throws an expectable exception. This is motivated by the fact, that

app developers usually omit redundant error handling code (i.e., "If this code executes, it means my app was installed and hence has been granted all requested permissions.") and thus throwing an unexpected security exception due to runtime permission revocation will result (most likely) in the 3rd party app crashing. For gracefully handling such revocation, we deploy two different strategies in our system:

Data-filtering. Filtering sensitive data from responses/callbacks to a querying app is always possible, when the app cannot make any presumptions about the data set size or values it is accessing. This inherently holds for ContentProviders, because the app cannot know if and what data is contained in the provider, but only in which format (i.e., Cursor) to expect the response. The simplest form of data-filtering is *data shadowing* as presented in, for instance, [20], which simply returns an empty Cursor or a Cursor with fake data to an app without access, thus emulating an empty or fake data set. A more sophisticated form of filtering, based on filtering rows, is presented in [11]. In our design, however, we filter *both* rows and columns from responses. Hereby, rows can be entirely removed [11], while for columns simply all table cells in this column are emptied in the response (in order to preserve the expected structure of the Cursor). Being able to filter both rows and columns is essential for a per-data access control, because rows represent entire entries (e.g., one contact), while columns represent the data of each entry (e.g., contact's address, phone number, etc).

Data filtering is also applied to certain system Services, for which a default or fake value can be safely returned. For instance, when polling the accelerometer data, a flat value can be returned to emulate motionlessness and thus effectively revoking the caller's permission to access this data. Similarly, location data or the IMEI can be filtered.

Expectable exceptions. Throwing an exception, which the app must be able to handle, because it is a potential event when calling a particular remote function, is applicable for gracefully controlling access to Services. For instance, when calling the LocationManager for high resolution location data, an exception is thrown if the device has no GPS capability. Thus, the calling app must be able to handle this exception. Throwing such an exception is hence also a means to gracefully revoke the app's permission to access the GPS location data even if there is a GPS module available.

4.3 Profile Server

Correctly configuring access control rules usually requires a certain expertise and awareness by the user. Since such prerequisites are seldom fulfilled by the common end-user, we provide in our architecture the means to import the access control rules for a specific app from an external source. In our setup, we leverage a *Policy Server*, to which users (or external experts) can upload privacy profiles for apps and from which end-users can download and deploy these policies to their phone. A reasonable alternative approach would be to allow app developers to ship their applications with a preconfigured privacy profile for their app, which is then automatically loaded during installation. This could encourage app developers to perform some form of self-control (possibly detecting unnecessarily requested permissions [28]) and could increase the end-user's trust in such apps.

4.4 Semantical Linking Filtering

Besides the problem of coarse-grained and irrevocable permissions, a second observation of how privacy sensitive user data can leak are semantic links between different providers/services. This means, that sensitive information can be retrieved from deputy providers or services instead of the provider/service responsible for management of this information.

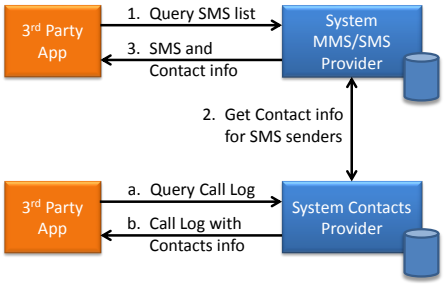


Figure 4: Semantic links between system SMS/MMS and ContactsProvider that allow inferring sensitive contacts information.

An actual example on a stock Android OS are the relations between the system SMS/MMS ContentProvider or CallLog provider and the system ContactsProvider (illustrated in Figure 4), through which a 3rd party app can infer sensitive contacts information – even if it has no direct access to the contacts information. In the first case (steps 1 to 3), an app with access to the SMS/MMS provider can query a list of sent and received SMS messages and their corresponding sender/receiver numbers. Thus, although this app might not have the permission to access the contacts provider directly (or the user prohibits this app to query contacts' phone numbers), it can infer from this result the user's contacts' phone numbers[2] and with more message parsing even information such as names or relation to the user (i.e., work, friend, etc.). Similarly, an app could retrieve the user's call log (steps a and b in Figure 4) and infer from it contacts' phone numbers. The CallLogProvider is actually part of the ContactsProvider app, but is addressed with a different URI (cf. Section 2.2) and requires a different permission (`READ_CALL_LOG`).

In our system, we address this issue by establishing automatic mappings between the access control rules of different providers/services, which reflect the semantic links between those providers/services. Figure 5 illustrates this mechanism with respect to the example in Figure 4. If the user configures in the Privacy Control App that an app, here *GO Contacts*, has no access to contacts phone numbers in the ContactsProvider, this decision is reflected in the access control rules of the MMS/SMS Provider by automatically prohibiting this app to access the SMS sender/receiver numbers. In our design, the Privacy Control App, since being in charge of the policy configuration and deployment, is responsible for applying this semantic links between the access matrices of different providers/services.

Technically, this semantic linking $\mathbb{SL}_{1,2}$ of access control rules is a mapping

$$\mathbb{SL}_{1,2}(s, o_1, d_1) \Longrightarrow s \times O_2 \mapsto d_2$$

[2]We assume that users most likely exchange SMS with people that they have in their address book.

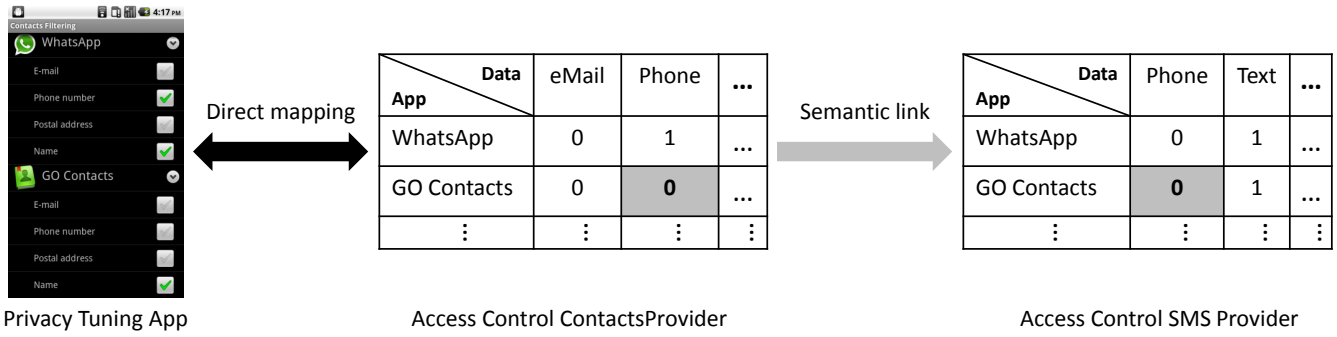


Figure 5: Mapping between policy configuration GUI and access matrix for ContactsProvider as well as semantic linking in case of ContactsProvider and SMSProvider (grey fields).

where s is the subject (i.e., app), o_1 is the object in the first provider/service, d_1 is the decision value in the first provider/service, O_2 are the objects in the second provider/service, d_2 is the new decision value in the second provider/service, and $\Longrightarrow$ denotes the operation to automatically deploy the right-hand side set of rules in the second provider/service.

In our architecture, we focus on the relations between *system* ContentProviders and Services, which are static for a particular version of the Android OS and in our experience usually also static across OS versions. Thus, these rules can be hardcoded in the Privacy Control App. Extending this mechanism to privileged 3rd party app providers and services, which also can act as deputies (e.g., if they export public interfaces to their internal information stores, in which they store sensitive information retrieved from the system providers/services) is more technical involved. It would require, for instance, instrumentation of those 3rd party apps [30] or taint-based access control on data flows [20] and we refer to future work for providing an efficient solution against this kind of *confused deputy* attack [19].

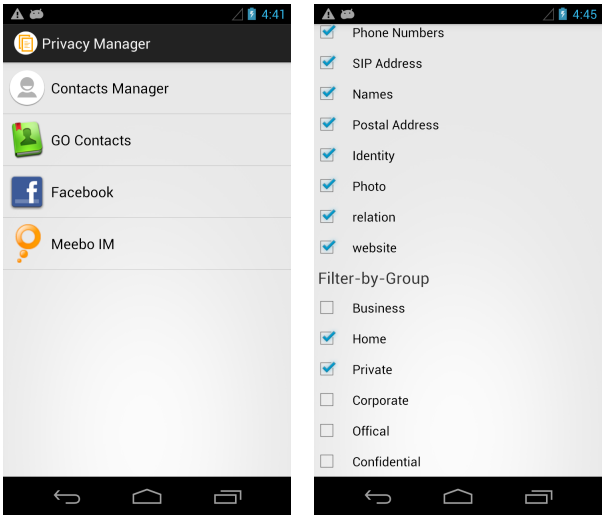
5. IMPLEMENTATION

We implemented our architecture for per-data access control for contacts and SMS/MMS data on both Android 2.2 (Froyo) and Android 4.1 (JellyBean).

5.1 Privacy Control App

Figure 6 depicts our Privacy Control App on Android 4.1 for configuring per-data access control to contact information. Figure 6(a) shows the list of all 3rd party apps in the system that hold the `READ_CONTACTS` permission and Figure 6(b) shows the access control configuration based on the mimetype of contacts data and the contacts groups. In this scenario, for instance, the app has access to all mimetypes of contacts that are members of the *Home* or *Private* group. Figure 5 contains a screenshot of the Privacy Control App on Android 2.2 and shows the mimetype-based access control for two 3rd party apps.

The Privacy Control App implements a broadcast receiver for the `PACKAGE_ADDED` and `PACKAGE_REMOVED` system broadcasts, upon which it initializes or removes, respectively, the access control rules for the added/removed app (cf. Section 4.2). For deploying new access control rules, we extend the system ContentProviders and Services with a corresponding interface (as we explain in the subsequent Sections 5.2 and 5.3).



(a) (b)

Figure 6: Screenshots of the Privacy Control App for configuring access control to the ContactsProvider: (a) list of installed 3rd party apps with `READ_CONTACTS` permission; (b) configuration of access control rules for a selected app based on mimetype (upper half) or groups (bottom half).

5.2 Filtering ContentProviders

To efficiently implement access control and filtering of private data stored in SQL tables in ContentProviders, we leverage mechanisms of the underlying SQL database.

Initialization. We store the access matrix as a new SQL table within the database for which we apply access control. This is a natural representation of access matrices, where each row represents one access control rule as defined in Section 4.2, containing the UID of a 3rd party app (i.e., the subject), the data mimetype and optionally an auxiliary value to further refine the mimetype (i.e., the object), and the access control decision (i.e., allow or deny). To add new access control rules for an app, identified by its sandbox' UID, the ContentProvider offers a new URI

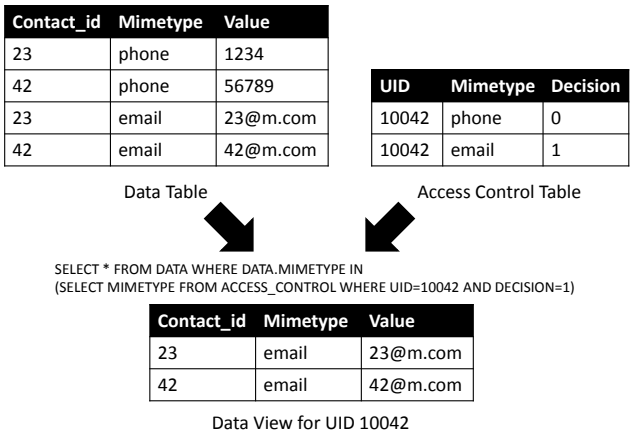


Figure 7: Example for combining an access control table and data table to construct an UID-specific View of the data table.

Listing 1: Concrete example for creation of SQL View for the Data table in Contacts Content-Provider

```
1  CREATE VIEW view_data_10043 AS SELECT * FROM
2  (SELECT * FROM view_data WHERE raw_contact_id IN
3  (SELECT raw_contact_id FROM view_data WHERE
       mimetype_id=11 AND data1 IN
4  (SELECT access_value FROM access_control WHERE
       app_uid=10043 AND mimetype_id=11))
5  OR raw_contact_id IN
6  (SELECT _id FROM raw_contacts WHERE _id NOT IN
7  (SELECT raw_contact_id FROM data WHERE
       mimetype_id=11)))
8  WHERE mimetype_id IN
9  (SELECT mimetype_id FROM access_control WHERE
       mimetype_id <> 11 AND
10 app_uid=10043 AND access_value=1) OR mimetype_id=11
```

`content://AUTHORITY/access_control` with authority, e.g., being `contacts` for the ContactsProvider. Upon access to this URI, the ContentProvider verifies with the help of Binder and the PackageManager that the caller is the trusted Privacy Control App or aborts operation otherwise. Through this interface, the Privacy Control App inserts the UID of the new app and the default decision value. Our new access control logic in the ContentProvider then inserts one new row for each available mimetype with the UID and the default permission value into the access control table.

Enforcement. Using SQL Views, we can efficiently establish representations of SQL tables, which realize data filtering. An SQL View is a virtual table, whose content is derived from queries to another table (or other View) and which can be used in SQL commands at all places a normal table can be used. In our system, we create one SQL View per app and per data table for which we want to realize access control. The SQL query to derive each View is constructed such, that it combines the data table and access control table by selecting from the data table only those rows, which are allowed according to the access control table. Figure 7 illustrates this mechanism and Listing 1 gives a concrete example for creating an SQL View for the

Listing 2: Concrete example for creation of an SQL Trigger to synchronize the access control table with the mimetypes table

```
1  CREATE TRIGGER mimetypes_aclinsert
2  AFTER INSERT ON mimetypes
3  WHEN NEW.mimetype<>
       "vnd.android.cursor.item/group_membership"
4  BEGIN
5  INSERT INTO access_control (app_uid, mimetype_id,
       access_value)
6  SELECT DISTINCT app_uid, NEW._id, 1
7  FROM access_control;
8  END
```

Data table in the ContentProvider for mimetype-based and group-based filtering of data for an app with UID 10043. To actually apply filtering, all queries by 3rd party apps are redirected to the corresponding SQL View for the app's UID, or to the default unfiltered table in case of queries by trusted system apps.

Updates. The new access control URI of the ContentProvider is further used by the Privacy Control App to query the current content of the access control table, e.g., in order to display it to the user, and also to update the access control table if the user performed any configuration.

Moreover, since the filtering is based on mimetypes and groups, the access control rules in the ContactsProvider must be kept in synchronization with the types and groups available. For instance, if a new group is added, a new access control rule must be created for all 3rd party apps and this new group, initially with the configured default value $\perp$. Similarly, if a group or mimetype is removed, all obsolete access control rules referring to this type/group must be deleted. To efficiently keep our access control table in sync with the available mimetypes and groups, we rely on SQL Triggers. Triggers execute specific SQL commands when particular events transpire on selected tables. In our system, we define Triggers on the mimetypes (cf. Listing 2) and groups tables of the ContactsProvider to perform the necessary synchronization operations.

5.3 Filtering Services

We extend system Services with new logic to perform access control enforcement of the form presented in Section 4.2. Hooks in the API functions that expose sensitive information redirect the control flow to this new access control logic. There, a per-data filtering is performed or an expectable exception is thrown, respectively. The underlying access control rules are stored in a local access vector cache.

To facilitate management (e.g., insertion, deletion, update) of access control rules in Services by the Privacy Control App, this new access control logic provides a new API function. This function is protected such, that only the trusted Privacy Control App or Android's SystemServer have permission to use it.

Initialization. For initializing the access control for a newly installed app, the Privacy Control App inserts the UID and the default decision value into the corresponding services. The access control logic then creates rules with this UID and decision value for each object (i.e., data) it manages. For instance, the LocationManager creates two

rules to regulate access to the coarse- and to the fine-grained location data.

Updates. To display the current access matrix of a service to the user, the Privacy Control App queries the current rules from the service. When the user adjusts the decision value of a rule, the updated rule is propagated back to the service.

Enforcement. Services implement different mechanisms, that allow an app to retrieve sensitive information. First, services offer API functions, that apps can explicitly call to get information (e.g., get last known or current location). As described above, we redirect the control flow of these functions to our access control logic and which filters data the calling app has no access to or throws an exception, respectively. Additionally, Services allow an app to receive sensitive information also via broadcast messages triggered by certain events (e.g., receipt of an SMS), through registered event listeners (e.g., change of location), or tokens (i.e., PendingIntent). Thus, to achieve correct coverage of our access control, we instrument the Service functions for callbacks to registered listeners and for sending PendingIntents to either filter data from responses or to suppress the response as a whole. To filter broadcast messages, we instrumented the broadcast subsystem of the ActivityManagerService similar to the approaches presented in [27, 11], however, basing the filtering on the access control rules retrieved from the sending service and any semantically linked service/provider.

6. EVALUATION

In this section, we provide the evaluation results for the performance and effectiveness of our implementation.

6.1 Performance

To evaluate the performance impact of our access control, we performed synthesized benchmark tests that emulate common access patterns to the system ContactsProvider and for SMS system notifications. Our test platform was a Samsung Galaxy Nexus (Maguro) with Android 4.1.1-r3 with and without our modifications.

6.1.1 System ContactsProvider

To evaluate the performance impact of our per-data access control to Contacts information, we deploy a Contacts database with 500 random Contacts, each Contact assigned random information such as name, instant messenger, phone number, or email address (in total 13 different data types per contact). These contacts are organized in six different, distinct groups (Group 1: 95 contacts; Group 2: 75 contacts; Group 3: 77 contacts; Group 4: 88 contacts; Group 5: 78 contacts; Group 6: 90 contacts). We devise the following test cases for our benchmark:

NumGroups: The benchmark app queries the number of existing groups by retrieving the *_id* column from the *Groups* URI. This tests the filtering for the Groups View.

NumContacts: The benchmark app queries the number of existing contacts by retrieving the *_id* column from the *Contacts* URI. This tests the group-based filtering of the Contacts View.

NumRawContacts: The benchmark app queries the number of existing raw contacts by retrieving the *_id* column from the *RawContacts* URI. This tests the group-based filtering of the RawContacts View.

RandomContactData: The benchmark app queries all data rows and columns for a randomly selected raw contact (the selection process is excluded from the measurement) from the *Data* URI. This tests the group-based and mimetype-based filtering of the Data View.

NumRandomGroup: The benchmark app queries the number of existing contacts that belong to a specific, randomly selected group by retrieving the *contact_id* column from the *Data* URI where the group membership value equals the selected group ID. This tests the group-based and mimetype-based filtering of the Data View.

Each test case is performed 1000 times in each of the following group filter settings for our benchmark app: *All Groups* (500 contacts available), *Only Group 1* (95 contacts available), *Only Groups 1, 4, and 6* (273 contacts available), and *Plain Android* (500 contacts available).

While usually all mimetypes were allowed, the test case *RandomContactData* was additionally performed with only the *Name* mimetype allowed for each group filter setting. This is motivated by the fact that this is the only test, where the mimetype-based filtering effectively influenced the performance, while all other tests purely rely on the speed of the group-based filtering.

Table 1 lists our performance measurements and Appendix A presents the corresponding Cumulative Frequency Distribution histograms. Relatively to the default Android, our nested `SELECT` statements in the filter Views impose a clear performance degradation. However, in absolute numbers, the overhead is still in a reasonable range when considering the usual frequency of queries and compared to the required time to process the query results on the caller side. Considering this, we argue that the performance impact does not negatively affect the user experience, even if several queries are performed subsequently.

6.1.2 SMS Notification

We further evaluate the performance impact of filtering system broadcast message about newly received SMS messages. The filtering is based on the phone number of the sender and whether the broadcast receivers have access to contacts phone numbers or the contacts group to which this numbers belongs. In 20 measurements, the average performance of creating the filtered receivers list with access control was 0.931ms. In default Android, we measured an average of 0.893ms for creating the receivers list, thus our access control add a negligible overhead of approximately 4%. However, after access control initialization/configuration, our architecture required a one-time overhead of 30ms on average to propagate the rules to the enforcement point in the Broadcast subsystem (cf. *Semantic Link* in Section 4.4).

6.2 Contacts Database Size

In our performance evaluation of the Contacts filtering, the contacts database containing 500 contacts, 13 data mimetypes, and 6 groups measured 2146304 bytes without any filtering views and 2150400 bytes with the views for one app. Thus, the filter views for one app adds 4096 bytes. This overhead depends purely on the number of defined mimetypes and groups for contacts data.

6.3 Effectiveness

To evaluate the effectiveness of our solution, we inserted the contact information of five real persons, who are also

Filtering All Groups Allowed		
Test	$\bar{t}$ (ms)	σ (ms)
NumGroups	6.302	7.594
NumContacts	32.705	9.080
NumRawContacts	26.719	7.445
RandomContactData	44.763 / 44.104	11.464 / 10.684
NumRandomGroup	28.036	7.035
Filtering Only Group 1 Allowed		
Test	$\bar{t}$ (ms)	σ (ms)
NumGroups	5.912	7.391
NumContacts	12.172	7.446
NumRawContacts	11.651	7.366
RandomContactData	30.926 / 30.216	12.208 / 11.184
NumRandomGroup	16.207	7.327
Filtering Only Groups 1, 4, 6 Allowed		
Test	$\bar{t}$ (ms)	σ (ms)
NumGroups	6.240	7.716
NumContacts	21.690	7.793
NumRawContacts	18.586	9.024
RandomContactData	38.576 / 38.090	10.657 / 11.847
NumRandomGroup	22.348	7.629
Default Android		
Test	$\bar{t}$ (ms)	σ (ms)
NumGroups	5.133	7.429
NumContacts	9.069	8.433
NumRawContacts	7.550	7.360
RandomContactData	17.888	9.244
NumRandomGroup	9.510	8.532

Table 1: Performance results for Contacts ContentProvider with average execution time ($\bar{t}$) and standard deviation (σ) in ms from 1000 executions each. For *RandomContactData* the first values represents filtering with all 13 mimetypes allowed and the second value with only the *Name* mimetype allowed.

WhatsApp users, into the contacts database. We successfully verified the effectiveness of our access control with WhatsApp v2.8.4313 (retrieved from the developer's website), GO Contacts v.146, Facebook app v1.9.1, and a synthesized test app for contacts management. The former two apps were chosen not only because of their popularity, but also because any filtering shows immediate effect. That is, when denying WhatsApp access to contacts phone numbers or all contacts groups, the app informs us about the fact, that none of our contacts is using the service. Similarly, GO Contacts presents only filtered data to which it has access. In all cases, the mimetype-based and group-based filtering of contacts data was successful and did not lead to an application crash.

As a side-effect of our tests, we could verify that the enumeration attack for WhatsApp presented in [29] is still applicable, since some of our randomly generated and unknown contacts where listed as our WhatsApp friends, indicating that the random phone number indeed belongs to a real person using WhatsApp. Moreover, we observed that the GO Contacts developers are aware of the logic in the ContactsProvider's SQL Command Factory (cf. Section 2.2) and designed their projection arguments to the Contacts Provider such, that they are able to embed custom subqueries into the default queries. We had to address this issue by implementing an additional sanity check on the arguments supplied by apps in order to also redirect the embedded subquery to the filtered SQL Views.

An open problem for future research are *URI Permissions* (c.f. Section 2.2). These could potentially be handled by creating ad-hoc access control rules based on additional user input by means of a popup dialog.

7. RELATED WORK

In the recent years, a number of security extensions to the Android OS have been proposed [30, 9, 21, 10, 11, 26, 13, 20, 25, 12, 27, 23, 32], which address privacy aspects of end-users.

Inlined Reference Monitors. To achieve policy enforcement for 3rd party apps on Android *without* the need to modify the operating system or to root the phone, some recent works leverage so-called *Inlined Reference Monitors* (IRM) [30, 9, 21]. IRM places the policy enforcement code directly in the 3rd party app instead of relying on a system centric solution. The implementations are based on runtime modification of the Global Offset Table (GOT) of the app process *(Aurasium)* [30] or modification of the Dalvik executable bytecode *(AppGuard, Dr. Android and Mr. Hide)* [9, 21].

An open problem with respect to filtering responses from privacy sensitive sources such as ContentProviders is the necessary statefulness of the reference monitor. To implement fine-grained filtering, it has to analyze and track the parameters of its host app to ContentProviders in order to correctly interpret and filter responses. For instance, if the query to the provider has a parameter which renames and reorders the columns of the response, the monitor has to be aware of this parameter in order to know which data is contained in which column of the response.

Android OS Security Extensions. The conceptually closest related work to our solution is the *TISSA* [32] architecture. In TISSA, the user can assign different trust levels to apps through a Privacy Setting Manager. Depending on its trust level, an app receives benign, fake, anonymized, or empty data when accessing privacy sensitive system ContentProviders or Services. Although the authors of TISSA acknowledge the problems of returning fake or empty data (cf. Section 3), they unfortunately omit to explain how they achieve anonymization of sensitive data. In contrast to TISSA, one design goal of solution was to explicitly allow per-data access control and, moreover, establishing semantic links between different enforcement points (Section 3.1), which otherwise might violate an access control like in TISSA.

The authors of *Porscha* [26] propose a DRM mechanism to enforce access control on specific data, such as SMS or e-mails, that is tagged with a DRM policy. This policy is attached to the data by the data creator. Porscha enforces this policy on data at rest (i.e., ContentProviders) as well as in transit (e.g., Intents) and denies non-policy-compliant apps access to the data. In contrast to Porscha, our solution focuses on *user-centric* policies to protect the user's privacy and considers a finer granularity of data (e.g., contacts phone number, email address). Moreover, our solution considers the links between providers/services over which data might leak. Thus, our solution could benefit Porscha by enabling a more fine-grained DRM and further preventing protected data from leaking through other system services/providers.

Similarly, frameworks such as *TaintDroid* [13] and the *AppFence* architecture [20], which builds on TaintDroid, track

the propagation of tainted data from sensible sources (in program variables, files, and IPC) on the phone and detect (or prevent, respectively) unauthorized leakage of this data. AppFence, additionally, provides data shadowing features for a variety of information sources (e.g., TelephonyManager, LocationProvider or ContactsProvider), i.e., it returns to apps empty or fake data. This per-interface access control can severely impede the app's user-desired functionality as stated in our problem description (Section 3). Both architectures could profit from our solution in order to provide a more accurate tainting, while our solution would benefit from taint tracking to address the confused deputy problem described in Section 4.4.

Both *APEX* [25] and *CRePE* [12] focus on enabling/disabling functionalities and enforcing runtime constraints. Both are related to our solution in the sense that they can restrict the capabilities of an app and protect data assets in shared resources like ContentProviders. Compared to our solution, the enforcement described in [25, 12] is, however, per-interface and provides only an *all-or-nothing* access. Similar to Porscha, their access control could profit from our solution to achieve more fine-grained runtime constraints.

Saint [27] introduces a context-aware access control model to enable developers to install policies to protect the interfaces of their applications. However, Saint only considers application developers as stakeholders in policy definition; our solution puts the privacy requirements of the end-user in the foreground. Moreover, developers in Saint could adopt our filtering mechanism for ContentProviders (cf. Section 5.2) to achieve a more fine-grained access control to their data.

Security Domains. Other related work aims specifically at establishing different security domains on Android [11, 10, 23] and addresses in this context (fine-grained) access control to information such as Contacts or SMS.

SE Android [23] is a port of SELinux [24] for the Android platform. It provides a kernel-level Mandatory Access Control (MAC) mechanism and is able to regulate Binder-based inter-process communication between apps. Thus, it is able to deny apps access to the system Services and ContentProviders. However, similar to [25, 12, 27], this is an *all-or-nothing* access to Services and ContentProviders.

The *TrustDroid* [11] architecture establishes two strictly isolated security domains on Android devices, one for private and one for business purposes. Similar to Porscha, it tags data, e.g., contacts, with a label, and filters (broadcast) Intents, but aims at preventing illegal data flows between the two domains. Using our per-data access control, TrustDroid could support more fine-grained policies.

The *XManDroid* [10] architecture aims at preventing confused deputy and collusion attacks by applying an access control framework similar to TrustDroid, but with more dynamic policies. Like TrustDroid, it could apply a more fine-grained access control on data, when adapting our solution.

8. CONCLUSION

In this paper, we presented a security architecture for Android that enables per-data and per-app access control to the user's private information in Android's system ContentProviders and Services. Due to the high resolution of our control, the user is empowered to adjust the access control rules (by means of a Privacy Control App) according to his individual privacy requirements and thus to satisfy both his need to share and his desire to protect his private information. A particular contribution of our work is the consideration of semantic links between different ContentProviders and Services by synchronizing the corresponding access control rules and hence preventing information inferring from deputy providers/services. In future work, we plan to extend our architecture to more use-cases and publicly release our implementation, since we believe that our work can be efficiently combined with related work such as taint-tracking [13] or Mandatory Access Control [23].

9. REFERENCES

[1] 3LM - Three Laws of Mobility. `http://www.3lm.com`.

[2] Facebook Caught Reading User SMS Messages? | TalkAndroid.com. `http://www.talkandroid.com/94623-facebook-caught-reading-user-sms-messages/`.

[3] National Institute of Standards and Technology, National Vulnerability Database, Vulnerability Summary for CVE-2009-1185. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1185`.

[4] National Institute of Standards and Technology, National Vulnerability Database, Vulnerability Summary for CVE-2011-1823. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1823`.

[5] Path uploads your entire iPhone address book to its servers. `http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html`.

[6] WhatsApp storing messages of users up to 30 days | Your Daily Mac. `http://www.yourdailymac.net/2012/02/whatsapp-storing-messages-of-users-up-to-30-days/`.

[7] WhatsApp took all my contacts and sent to their servers without asking me - BlackBerry Forums at CrackBerry.com. `http://forums.crackberry.com/blackberry-apps-f35/whatsapp-took-all-my-contacts-sent-their-servers-without-asking-me-649363/`.

[8] Whisper Systems. `http://www.whispersys.com/permissions.html`.

[9] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard - real-time policy enforcement for third-party applications. Technical Report A/02/2012, Max Planck Institute for Software Systems, 2012.

[10] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS'12)*, 2012.

[11] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, 2011.

[12] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC'10)*, 2010.

[13] W. Enck, P. Gilbert, 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. In *9th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI'10), 2010.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *20th USENIX Security Symposium*, 2011.

[15] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy Magazine*, 2009.

[16] Google. The Android developer's guide - Android Manifest permissions. `http://developer.android.com/reference/android/Manifest.permission.html`, 2012.

[17] Google. Content providers. `http://developer.android.com/guide/topics/providers/content-providers.html`, 2012.

[18] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)*. ACM, 2012.

[19] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

[20] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM conference on Computer and communications security (CCS'11)*. ACM, 2011.

[21] J. Jeon, K. K. Micinski, J. A. Vaughan, N. Reddy, Y. Zhu, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained security policies on unmodified android. Technical Report CS-TR-5006, University of Maryland, Department of Computer Science, 2012.

[22] Lookout Mobile Security. Security alert: Geinimi, sophisticated new Android Trojan found in wild. `http://blog.mylookout.com/2010/12/geinimi_trojan/`, 2010.

[23] National Security Agency. Security Enhanced Android. `http://selinuxproject.org/page/SEAndroid`.

[24] National Security Agency. Security-Enhanced Linux. `http://www.nsa.gov/research/selinux`.

[25] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*, 2010.

[26] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in Android. In *26th Annual Computer Security Applications Conference (ACSAC'10)*, 2010.

[27] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *25th Annual Computer Security Applications Conference (ACSAC'09)*, 2009.

[28] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *18th ACM conference on Computer and communications security (CCS'18)*. ACM, 2011.

[29] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. Weippl. Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In *19th Annual Network & Distributed System Security Symposium (NDSS'12)*, 2012.

[30] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *21st USENIX Security Symposium*. USENIX, 2012.

[31] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95 –109, may 2012.

[32] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *4th international conference on Trust and trustworthy computing (TRUST'11)*. Springer-Verlag, 2011.

APPENDIX

A. CUMULATIVE FREQUENCY DISTRIBUTION FOR CONTACTS BENCHMARK TESTS

For better readability, all histograms show only the time thresholds for which the cumulative frequency of each test is 99.90%.

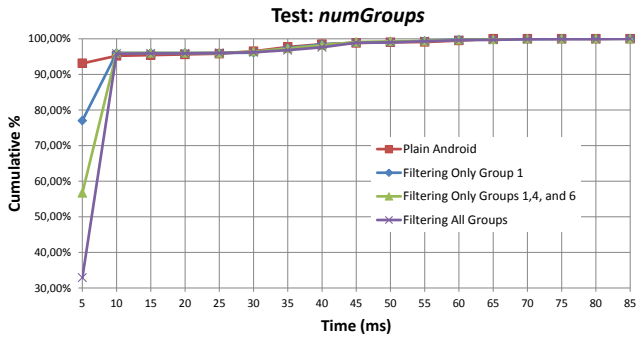


Figure 8: Cumulative Frequency Distribution of test NumGroups.

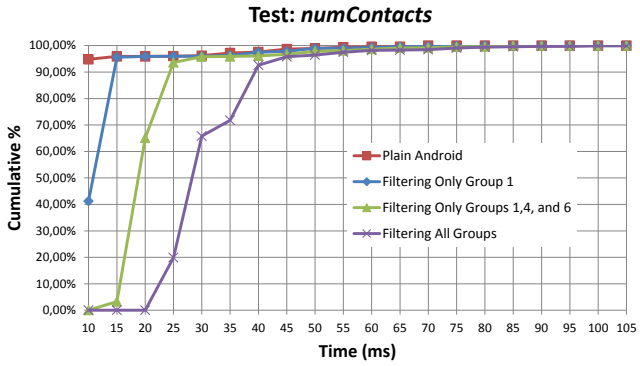


Figure 9: Cumulative Frequency Distribution of test NumContacts.

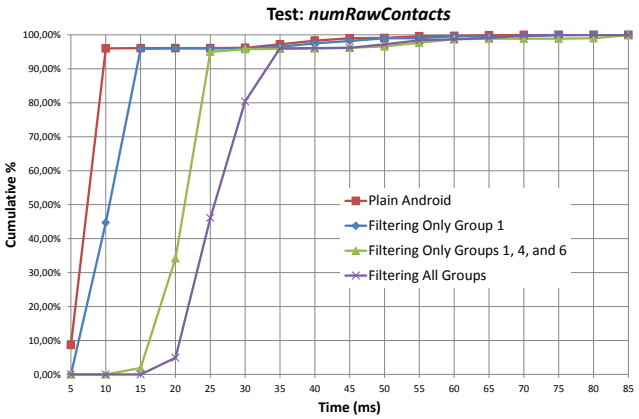


Figure 10: Cumulative Frequency Distribution of test NumRawContacts.

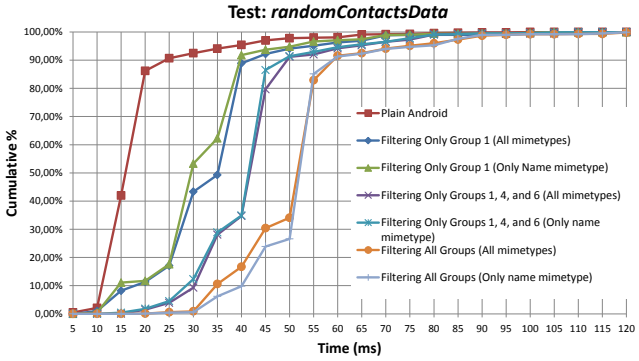


Figure 11: Cumulative Frequency Distribution of test RandomContactsData.

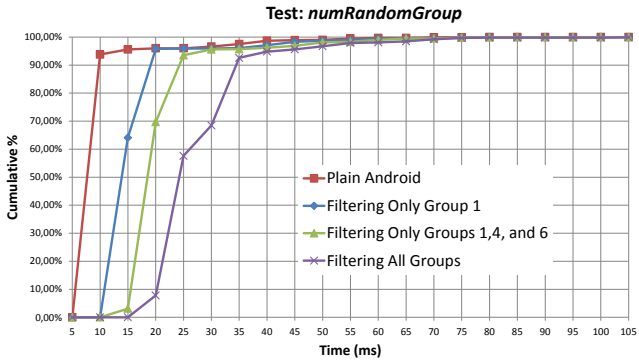


Figure 12: Cumulative Frequency Distribution of test NumRandomGroup.