Mechanisms for the Dynamic Installation and Control of Data Collection Tasks on Smartphones

PRESENTED THE October 13, 2015

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF THESSALY

FOR GRADUATION OF MASTER OF SCIENCE

BY

Emmanouil KATSOMALLOS

supervised by:

Spyros Lalis, Associate Professor, University of Thessaly
Thanasis Papaioannou, Senior Researcher, CERTH
George Theodorakopoulos, Lecturer, Cardiff University

Greece
2015

Acknowledgements

Upon the completion of my thesis, I would like to express my deep gratitude to my research supervisor Dr. Spyros Lalis for his patient guidance, enthusiastic encouragement and useful critiques of this research work. Besides my advisor, I would like to thank the rest of my thesis committee, Dr. Thanasis Papaioannou and Dr. George Theodorakopoulos for their invaluable contribution.

A special thanks to my department's faculty, staff and fellow students for their valuable assistance whenever needed and for creating a pleasant and creative environment during my studies.

Last but not least, I wish to thank my family and friends for their unconditional support and encouragement all these years.

Volos, October 13, 2015

Περίληψη

Η διατήρηση της ιδιωτικότητας του χρήστη είναι ζωτικής σημασίας για την ευρεία υιοθέτηση εφαρμογών ανίχνευσης χρήστη και εθελοντικής ανίχνευσης που βασίζονται σε προσωπικές συσκευές. Επί του παρόντος, κάθε εφαρμογή έχει τη δική της καλωδιωμένη και ενδεχομένως ατεκμηρίωτη υποστήριξη ιδιωτικότητας (εάν υπάρχει), ενώ οι οριζόντιοι μηχανισμοί προστασίας που παρέχονται από τα λειτουργικά συστήματα και συστήματα εκτέλεσης λειτουργούν σε χαμηλό επίπεδο που μπορούν να βλάψουν σημαντικά τη χρησιμότητα της εφαρμογής, ή ακόμα να καταστήσουν μια εφαρμογή άχρηστη. Για να επιτευχθεί μεγαλύτερη ευελιξία, προτείνουμε ένα πλαίσιο για την αποσύνδεση του μηχανισμού ιδιωτικότητας από τη λογική της εφαρμογής, ώστε να μπορεί να αναπτυχθεί από ένα άλλο, ίσως πιο αξιόπιστο πρόσωπο, και επιτρέποντας το δυναμικό δέσιμο διαφορετικών μηχανισμών ιδιωτικότητας με την ίδια εφαρμογή πριν τα δεδομένα αφήσουν τη συσκευή για το Διαδίκτυο. Συζητάμε μια απόδειξη της εφαρμογής του προτεινόμενου πλαισίου για Android, όπου μηχανισμοί ιδιωτικότητας αναπτύσσονται ανεξάρτητα ως ξεχωριστά συνδεόμενα μέρη, χρησιμοποιώντας μια απλή διεπαφή εφαρμογής προγράμματος με σαφή υποστήριξη για συλλογικά σχήματα που περιλαμβάνουν περισσότερες από μία προσωπικές συσκευές. Αξιολογούμε επίσης το φόρτο επεξεργαστή, την δραστηριότητα δικτύου, τη χρήση μνήμης και την κατανάλωση μπαταρίας της εφαρμογής μας.

Λέξεις-κλειδιά

πλαίσια ιδιωτικότητας; μηχανισμοί ιδιωτικότητας; ανίχνευση πλήθους; εθελοντική ανίχνευση; έξυπνα τηλέφωνα

Abstract

Preserving user privacy is crucial for the wide adoption of crowdsensing and participatory sensing applications that rely on personal devices. Currently, each application comes with its own hardwired and possibly undocumented privacy support (if any), while the horizontal protection mechanisms provided by operating and runtime systems operate at a low level that can significantly harm application utility, or even render an application useless. To achieve greater flexibility, we propose a framework for decoupling the privacy mechanism from the application logic, so that it can be developed by another, perhaps more trusted party, and for allowing the dynamic binding of different privacy mechanisms to the same application before data leaves the device for the Internet. We discuss a proof-of-concept implementation of the proposed framework for Android, where privacy mechanisms are independently developed as separate plug-in components, using a simple API with explicit support for collaborative schemes that involve more than one personal devices. We also evaluate the CPU load, network activity, memory usage and battery consumption of our implementation.

Keywords

privacy frameworks; privacy mechanisms; crowdsensing; participatory sensing; smartphones

Contents

List of Figures

List of Tables

List of Listings

Chapter 1 Introduction

According to a recent Gartner report [10], smartphone sales in 2014 grew 29% vs. 2013, corresponding to roughly 2/3 of the total sales of mobile devices, at an estimated 1.2 billion users worldwide. Given this wide adoption, and being carried by most people around the clock everywhere they go, the smartphone is the most ubiquitous sensing device on the planet, and a key enabler for crowdsensing applications. It comes with various onboard sensors, such as GPS, front/backside cameras, accelerometers and microphone. Moreover, it can serve as a hub for an even broader range of sensors embedded in clothes [15] and wearable artifacts [16], as well as other personal mobile objects like a bicycle [13].

However, to realize the vision of mass-scale crowdsensing based on mobile personal devices, like the smartphone, several challenges need to be tackled. For instance, one has to deal with asymmetrical and intermittent connectivity, save precious battery lifetime, simplify the installation and management of sensing applications on the smartphone, and address various privacy and trust concerns. In fact, privacy turns out to be of key importance so that people indeed agree/volunteer to provide data via their mobile personal devices, and can be a show-stopper for many crowdsensing efforts[1,2].

The privacy issue has already gathered a lot of attention among researchers, and different methods for preserving user privacy in crowdsensing scenarios have been proposed and studied in the literature. But few of these methods are adopted in practice. These implementations are also tightly-coupled with the application logic, making them hard to be inspected, replaced, let alone reused in other applications. Basically, if people wish to participate in a data collection campaign, they have to take the corresponding mobile application "as is", bundled with whichever privacy mechanism (if any) was considered to be appropriate by the developer. There is also a conflict of interest here: strong privacy is desirable for the user but will typically lead to reduced utility for the application, so the application developer has no direct incentive to integrate strong privacy support into the application. And even if the application developer has good intentions, she may not be a privacy expert, hence may be unaware of better privacy methods that could be used in conjunction with the application. Finally, each user may have different privacy concerns, requiring different privacy mechanisms.

Motivated by the above observations, we propose an open framework for the flexible development, installation and activation of different privacy mechanisms on mobile personal devices, as independently developed software components that can be used in conjunction with sensing applications that run on the device. We also describe a prototype implementation of this framework for *EasyHarvest* [12], a crowdsensing system for Android smartphones and tablets, and illustrate its flexibility by showing how both standalone and collaborative mechanisms can be supported for the same crowdsensing application.

The main features of our privacy framework are briefly as follows: (i) Privacy mechanisms are developed using a simple yet also quite powerful API, which provides explicit support for collaborative schemes. Concretely, a privacy mechanism can receive updates on the presence of peer devices that run the same mechanism in order to adapt its operation accordingly and exchange data with its peers. (ii) Privacy mechanisms are uploaded to a public repository and can be freely inspected by the community. The user can then pick and install the privacy mechanism that seems most appropriate for her needs. (iii) On the personal device, privacy mechanisms are dynamically linked to the mobile application that produces the data. (iv) The user can adjust the privacy level of the mechanism, or change the privacy mechanism for a given application, as desired.

Most proposed privacy frameworks for mobile phones in effect block access to various sensor and personal data feeds for selected or all applications. These frameworks adopt a low-level binary "on/off" approach for individual data feeds, before they reach the application. But once the application is given access to these data feeds, there is no control on the data the leaves the device, and it is left for the user to assess the risks of such an information exposure. Furthermore, blocking data feeds at a low level may greatly diminish or even nullify the functionality of the application. In contrast, our framework can support more complex and flexible privacy

[1] www.pcworld.com/article/261935/survey_mobile_users_care_about_data_privacy.html

[2] www.net-security.org/secworld.php?id=18154

mechanisms, allows for higher data utility for the application, and removes a significant decision-making burden from the user who does not have to trust the application developer.

The rest of the thesis is structured as follows. 0 introduces the proposed concept and high-level system model. 0 gives an overview of the *EasyHarvest* system. 0 describes a proof-of-concept implementation of the proposed privacy framework for it, and provides examples of simple standalone and collaborative privacy mechanisms. Chapter 5 evaluates the privacy framework in terms of the overhead that is introduced on top of the basic *EasyHarvest* system. Chapter 6 gives an overview of related work. Finally, Chapter 7 concludes the thesis and identifies directions for future work.

Chapter 2 Conceptual Approach

Users who provide data to crowdsensing applications via their personal devices have several privacy aspects that might need to be hidden, e.g., user identity (linking attacks), user location (trajectory tracing), user activities (activity tracing), or sensitive attributes (eavesdropping). Different mechanisms have been proposed to tackle the different threats [4], e.g., using data hiding (suppression), perturbation (adding noise to the data or adding fake data), obfuscation (generalization, mixing) and anonymization. These mechanisms may be collaborative (involve multiple users) or standalone, and they may be internal to the user devices producing the data or rely on external trusted third parties for privacy preservation. Also, mechanisms differ in their effectiveness on protecting the user against the various privacy threats, as well in their impact on the data utility for the various applications.

Despite this plethora of privacy mechanisms, crowdsensing applications come with hardwired privacy protection support, if any. Even if the code is open for inspection, the average user clearly does not have the expertise or the time to check the existence and/or effectiveness of the privacy mechanism of every single application. As a consequence, in practice, the user simply has to trust the persons/organizations that develop and manage these applications.

We believe that this problem can be addressed, to a large extent, by introducing privacy support for crowdsensing applications based on the following principles:

- **Decoupling.** Separate the sensing part of the application which runs on the personal device and generates data based on local sensor and personal data feeds, from the mechanism that preserves the privacy of the user who contributes the data. Ideally, the sensing part of the application should be developed without any concern for user privacy.

- **Diversity/Flexibility.** Support the development of privacy mechanisms which may have different characteristics and may protect different privacy attributes. In particular, provide support for collaborative privacy schemes, which may involve interaction between multiple personal devices and adapt their behavior accordingly.

- **Locality.** Preserve privacy locally, on the personal device or among a group of trusted devices. Once data leaves the personal device and reaches external components of the crowdsensing application on the Internet/cloud, the user has practically no control over it. Also, support collaborative privacy mechanisms while minimizing or even eliminating the reliance on centralized trusted third parties.

- **Utility.** Place the privacy mechanisms at a proper stage of the information production pipeline, so that it can intercept and process/filter privacy-sensitive data without completely destroying its utility. Enable the association of each application with the most suitable privacy mechanism – one that can effectively protect one or more privacy aspects that are of importance to the user, while at the same time preserving application utility.

Followingly, we propose an approach for the structured development and deployment of crowdsensing applications and privacy mechanisms on mobile personal devices, shown in Figure 1. On the one hand, the application owner declares the data model of the application, and provides the sensing component for the personal device. The data model defines the data items and allowed values that are produced by the sensing component that runs on the personal device – it may also include additional information about the transformations that can be performed on the data without harming its utility for the application. On the other hand, privacy experts review this data model, and provide suitable privacy mechanisms for it, which may filter, distort or aggregate data to protect certain privacy attributes while maintaining data utility for the application in question. The type of the transformation performed by a given privacy mechanism, and the impact on data utility can be described via suitable (human and machine-readable) metadata. Finally,

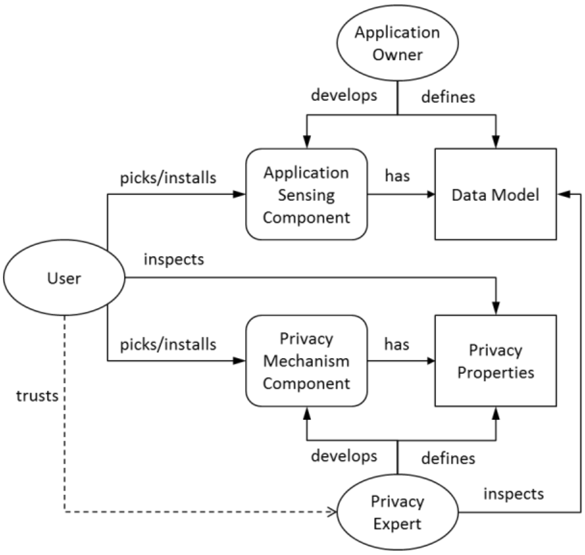

Figure 1. Towards a structured introduction of privacy mechanisms for crowdsensing applications: stakeholders and software artifacts.

the user who wishes to contribute to a crowdsensing effort picks and installs on her personal device the respective mobile sensing component along with one or more compatible privacy mechanism components.

On the personal device, a suitable runtime system supports the dynamic installation, binding and execution of these software components. Figure 2 shows an indicative high-level architecture. In a nutshell, the data produced by the sensing component of the application is fed into the privacy mechanism component, which in turn outputs the same type of data towards external application components (these will typically reside on a remote computing infrastructure, to perform data aggregation, processing and visualization). From a purely functional perspective, the privacy mechanism is transparent to the application (although the transformation performed on the data may affect utility). This way it becomes possible to use different privacy mechanisms for the same application. It is also possible to change the privacy mechanism for a given application on the fly, with the runtime taking care of the respective re-binding behind the scenes. Such a change can be requested by the user, or even be performed automatically by the runtime. Of course, the application can still work without any privacy component bound to it: for instance, the owner of the personal device may not care about privacy, or may fully trust the owner of a specific crowdsensing application.

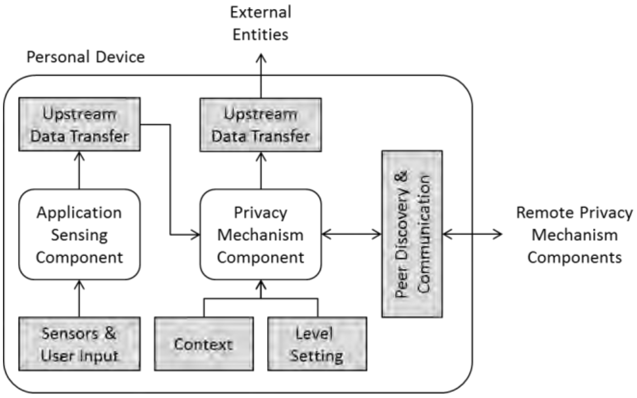
External Entities
Personal Device
Upstream Data Transfer
Upstream Data Transfer
Application Sensing Component
Privacy Mechanism Component
Peer Discovery & Communication
Remote Privacy Mechanism Components
Sensors & User Input
Context
Level Setting

Figure 2. High-level system architecture of the runtime system on the personal device. Grey boxes stand for the system interfaces/hooks.

The runtime system is responsible for providing the interfaces/hooks needed for these components to implement their functionality. More specifically, the sensing component should be given access to the local onboard sensors and other personal data feeds (possibly including explicit user input) in order to produce data that is required for the crowdsensing application. It also needs an interface for sending the produced data upstream (which the runtime redirects to the privacy mechanism component). Similarly, the privacy mechanism component needs to receive/retrieve context information to adjust its operation accordingly, and should be given the means to interact with peer components on other devices to be able to work in a collaborative way. Further, it should be possible to change the desired level of the privacy mechanism, triggering corresponding adjustments in the type and/or magnitude of the data transformation that is performed internally.

The main advantages of this approach are the following:

First of all, there is no attempt to control the type and/or amount of data that is consumed by the sensing component of the application at a low level. The application component can freely access the sensors and personal data feeds of the personal device (subject to the restrictions of the runtime API and low-level access control mechanisms). The information gathered by the application sensing component on the personal device is considered harmless, as long as it does not propagate to the Internet. However, precisely this information flow is controlled, through the privacy mechanism component and the level at which this is set to operate, as chosen by the user.

Secondly, the privacy mechanisms for a given application can be developed exclusively based on the corresponding data model, without having access to the code of the sensing component or any other external component of the application. Of course, the application must declare the data model truthfully. But doing so is in its own interest. More specifically, false declarations concerning the value ranges and utility of the data will result to the application being associated with an inappropriate privacy mechanism that will partly destroy or completely invalidate the data that is being produced. Also note that the runtime system can check the data produced by the application sensing component for type and value range compliance, and terminate/blacklist applications that violate their officially declared data model.

Thirdly, the user places her trust in the expert who develops the privacy mechanism, rather than in the owner of the crowdsensing application (of course, the runtime system on the personal device has to be trusted too). This greatly reduces the risk of privacy leaks since the privacy expert has no incentive to cheat the user. Moreover, the trust in a given mechanism can be enhanced by letting different experts review and certify the method and its concrete implementation (one can reasonably assume that the developer of a privacy mechanism will gladly make the code available for inspection). Privacy mechanism components can also be digitally signed so that the runtime system can verify their integrity before installing them on the personal device.

One question that arises is if someone will be motivated to contribute a privacy mechanism for a crowdsensing application that was developed and is run by others. There are good reasons to believe so. On the one hand, privacy experts can see this as a challenge and participate voluntarily, by developing new privacy mechanisms, as well as by reviewing or enhancing existing ones, in the spirit of many other thriving open source communities. On the other hand, given the clear decoupling between the core sensing part of the application and the mechanism that protects user privacy, owners of crowdsensing applications that need mandatory access to several onboard sensors and personal data feeds, will have a strong interest in their applications being linked to modular and easy-to-inspect privacy mechanisms, in order to boost the application’s popularity, and thus could pro-actively invite privacy experts to develop suitable privacy mechanisms for it.

Chapter 3 Overview of the EasyHarverst System

The *EasyHarvest* system was designed to simplify the development, managed deployment and controlled execution of sensing tasks on personal devices. Each sensing task is an application-specific agent to be replicated on a large number of devices. Focus is on background tasks that use the sensors of personal devices, without any explicit user input (though the system could be extended to support this), and produce data over a longer time period, in a best-effort manner. Besides taking sensor measurements, a task can perform custom processing on the device, before uploading data to the Internet.

EasyHarvest follows a client-server architecture, shown in the lower part of Figure 3. The server is managed by the community/organization that wishes to support crowdsensing applications. Application owners submit sensing tasks to the server, which in turn automatically deploys them on personal devices and collects the data produced by them. At any point, one can inspect the deployment progress of a given task, and retrieve the data collected so far. One can also suspend, resume or permanently remove a task from the server.

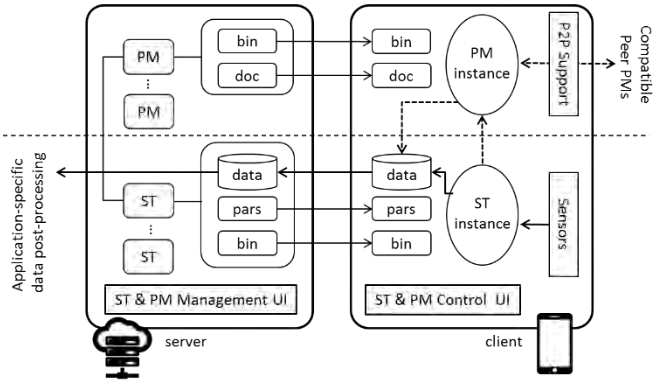

Figure 3. High-level architecture of *EasyHarvest*. Sensing tasks (STs) are retrieved from the server and are executed by the client on the smartphone. The extensions made to support flexible plug-in privacy mechanisms (PMs) are shown above the dashed horizontal line. The data flow from the ST to the server and application back-end is in bold lines; dashed lines illustrate this data flow when using a PM.

The *EasyHarvest* client provides the runtime environment for sensing tasks on the personal device. It can subscribe to one or more servers, which it inquires about application tasks that need to be executed. If the client decides to accept a task, it downloads the binary, creates a new task instance locally, and schedules it for execution on the device. The data produced by the sensing task is transferred to the server behind the scenes while dealing with disconnections in a transparent way.

Sensing tasks can be associated with a target geographical location and time period within the day. These parameters are supplied when the task is submitted to the server and can be modified later, if desired. The client receives these parameters together with the task binary, occasionally checks the server for updates, and accordingly activates or deactivates the sensing task depending on the current location and time context of the device. Post-processing of the data collected on the server is left for external, application-specific subsystems, which can retrieve this data via a suitable machine interface.

The current *EasyHarvest* prototype is designed for Android devices. The client is a user-level application that interacts with the server via a REST-based interface. Sensing tasks have the form of Java classes for the Dalvik environment[3], and implement a predefined

[3] source.android.com/devices/tech/dalvik/

interface though which they interact with and are controlled by the client runtime. The main primitives of the task interface are summarized in Table 1.

Primitive	Description

`void onStart(`
`  Context c,`
`  ObjectInputStream s`
`);`
	Initialize task in order to (re)start execution; previously saved state can be retrieved via the input stream.

`void onStop(`
`  ObjectOutputStream s`
`);`
	Release resources held by the task; optionally save state in the output stream.

`List<Object> getData();`	Return new data produced by the task since the last invocation.

Table 1. Sensing task interface.

Sensing tasks access the sensors (and possibly other data sources) of the smartphone via the native Android API. Note that the application developer is free to define the data objects that will be produced by the task. These must be serializable so that they can be transferred over the network, and may not contain any private fields. Respective checks are made when a task is submitted to the *EasyHarvest* server, as part of the process for producing the final binary. Also, at runtime, the *EasyHarvest* client performs type-checks to verify that the task indeed produces the expected type of data.

As an example, assume we wish to find the most crowded places in a city, over different periods of the day. For this purpose, one could employ the sensing task shown in Listing 1.

```
public class HotSpotData implements Serializable  {
        public Location loc; // geographical location

        public HotSpotDetectionData(Location loc) {
               this.loc = loc;
        }
        … // custom serialization methods, as needed
};

public class HotSpotDetector implements LocationListener {
        List<HotSpotData> data;
        Location loc;

        public void onStart(Context c, InputStream s) {
               data = new ArrayList<>();
                …
               timer = new CountDownTimer(…) {
                       …
                       public void onFinish() {
                               data.add(new HotSpotData(loc));
                       }
               };
               …
               locMgr = (LocationManager) c.getSystemService(…);
               locMgr.requestLocationUpdates(…, this);
        }

        void boolean onStop(OutputStream s) {
               timer.cancel();
               locMgr.removeUpdates(this);
        }

        public List<Object> getData() {
               List<HotSpotData> tmp = data;
               data.clear();
               return tmp;
        }

        public void onLocationChanged(Location loc) {
               this.loc = loc;
               timer.cancel();
               timer.start();
        }
}
```

Listing 1. Sensing task that records a person’s hotspots.

The above task employs the location sensor and a timer to infer how long the user remains at the current location. If the duration is above a threshold, the location is added to a list of hotspots, which is sent upstream, to the server.

The user can configure the *EasyHarvest* client to contact one or more servers for task download, and to ask for explicit permission before accepting a sensing task. One can also set the desired client load (the frequency at which the client runtime polls the sensing task for data, and synchronizes with the server) as well as the type of connectivity to use for the communication (Wi-Fi, cellular). Last but not least, it is possible for the user to define so-called privacy regions (in time and space) where all sensing tasks are suspended.

We note that in our current prototype the sensing task can freely access the resources of the smartphone via the Android API. Safer and more controlled task execution can be achieved by adopting existing sandboxing techniques [1]. Another limitation (mainly for debugging purposes) is that the client will accept/run only one sensing task at a time.

Chapter 4 The EasyHarvest Privacy Framework

We extended the *EasyHarvest* system to include support for flexible plug-in privacy mechanisms, along the lines of the conceptual approach described in Section II. The key elements of the extended system architecture are shown in Figure 3. Next, we give an overview of the implementation, as well as examples of standalone and collaborative privacy mechanisms for the sensing task that was discussed earlier.

4.1 Privacy mechanism registration and installation

Similar to application sensing tasks, privacy mechanisms are implemented as independent software components which are registered with the *EasyHarvest* server. Using a web-based interface, the privacy expert uploads the source code, associates the privacy mechanism with one or more sensing tasks, and provides a description of the mechanism and its privacy-preservation properties. The server compiles the code, and checks it for the required methods (see next section).

As part of its periodic interaction with the server, the *EasyHarvest* client queries about privacy mechanisms that can be used for the application sensing task that runs locally on the smartphone – this can be done in the background or at the user's request. The user browses the list of suitable privacy mechanisms, and selects the one to employ for the application. In turn, the client downloads the privacy mechanism on the phone, and instantiates/binds it to the sensing task. At runtime, the user can change the privacy level, completely deactivate the privacy mechanism, or select/switch to another privacy mechanism, as desired.

Method / Data Primitives	Description

`void onStart(`
`  Context c,`
`  int privLevel,`
`  ObjectInputStream s`
`);`
	Initialize the privacy mechanism, for the supplied privacy level; previously saved state can be retrieved via the input stream.

`void onStop(`
`  ObjectOutputStream s`
`);`
	Release resources held by the privacy mechanism; optionally save state in the output stream.

`List<PMData>`
`handleAppData(`
`  List<Object> data`
`);`
	Process the data produced by the sensing task, return the data to send upstream.

`List<PMData> handle-`
`PeerData(`
`  List<PMData> data`
`);`
	Process the data received from a peer, return the data to send upstream. (only for collaborative privacy mechanisms)

`void onLevelUpdate(`
`  int privLevel`
`);`
	Adjust internal operation based on the newly supplied privacy level setting.

`void onPeerGroupUpdate(`
`  List<PeerInfo> peers`
`);`
	Adjust internal operation based on the updated peer group configuration. (only for collaborative privacy mechanisms)

`public class PMData {`
`  int destID;`
`  List<Object> data;`
`}`
	Data structure for the data produced by the privacy mechanism, along with the identifier of the destination for this data (0 for the server; <>0 for a peer).

`public class PeerInfo {`
`  int peerID;`
`  int privLevel;`
`}`
	Data structure for peer information, consisting of the peer identifier and its current privacy level.

Table 2. Privacy mechanism interface.

4.2 Interface of privacy mechanism components

In terms of software development, privacy mechanisms are implemented as Java classes for the Dalvik environment. By convention, a privacy mechanism shall provide a pre-defined interface, summarized in Table 2. Note that only a subset of this interface is mandatory (highlighted rows of the table).

The data interface of privacy mechanisms refers to abstract data objects, just as this is the case for application sensing tasks. However, the data that will be actually forwarded to a privacy mechanism at runtime depends on the sensing task to which the mechanism will be bound. As already discussed in Section II, the developer of the privacy mechanism must be familiar with the application-specific data produced by the sensing task, in order to handle it properly. Also recall that the privacy mechanism is expected to generate the same type of data towards the server – corresponding type checks are done by the client runtime, before sending the data upstream.

4.3 Flexible privacy schemes

The above interface allows for the development of fully standalone as well as collaborative privacy mechanisms. It is also possible to implement privacy mechanisms that adapt their mode of operation, switching between standalone and collaborative mode, depending on the presence of other peers.

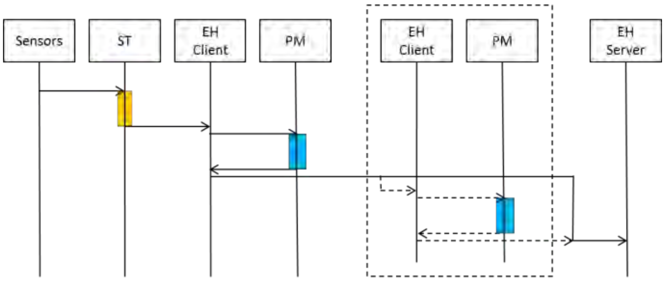

Figure 4. Data flow for standalone (solid lines) and collaborative privacy mechanisms (dashed lines). Data processing at the level of the sensing task (ST) and the privacy mechanism (PM) is denoted by the orange and blue vertical bars, respectively.

Standalone privacy mechanisms do not provide the `handlePeerData()` and `onPeerGroupUpdate()` methods. When the *EasyHarvest* client detects that these methods are missing, it hardwires the privacy mechanism to work in isolation, as illustrated in Figure 4 (solid lines). Also, in this case `handleAppData()` should always set the destination of the returned `PMData` object to zero, indicating to the client that the data produced should be sent directly to the server. As an example, Listing 2 shows the relevant code excerpts for a simple implementation which adds noise to the location component of the data that is produced by the sensing task discussed in the previous section (Listing 1).

```
public class SimpleStandalonePM {
        int level;
        List<PMData> buf;

        public void onStart (Context c, int privLevel, InputStream s) {
               buf = new ArrayList<>();
               level = privLevel;
        }

        public void onStop (OutputStream s) {}

        public List<PMData> handleAppData (List<Object> data) {
               for (int i = 0; i < data.size(); i++) {
                       data.set(i, distortLocation((HotSpotData) data.get(i), level));
               }
               buf.get(0).destID = 0; // send to server
               buf.get(0).data = data;
               List<PMData> tmp = buf;
               buf.clear();
               return tmp;
        }

        public void onLevelUpdate (int privLlevel)
               level = privLevel;
        }
}
```

Listing 2. Simple standalone privacy mechanism for the hotspot sensing task, which adds some noise to the location information.

Collaborative privacy mechanisms have to implement all the methods of the interface, depending on the desired functionality. More specifically, `handleAppData()` is used to forward locally generated application data to other peers for further processing and aggregation. Data arriving from one or more peers is processed via the `handlePeerData()` method. Note that it is again possible for the returned data to be sent to another peer or the server, depending on the destination of the `PMData` object. A typical information flow pattern is for every peer to process the application data, and then forward it to a designated peer, which can further process the data before sending it to the server, as illustrated in Figure 4 (dashed lines). An indicative implementation that works along this line is given in Listing 3, for the same application sensing task as above. In this example, the peer with the smallest identifier performs a simple form of anonymization, by bundling locally produced data with the data that is received from other peers, and sending the entire bundle to the server. As a result, the server does not know the original producer(s) of this data.

```
public class SimpleCollaborativePM {
        List<PMData> buf1,buf2;
        int aggrID, level;

        public void onStart(Context c, int privLevel, InputStream s) {
               buf1 = new ArrayList<>();
               buf2 = new ArrayList<>();
               aggrID = 0; level = privLevel;
        }

        public void onStop (OutputStream s) {}

        public List<PMData> handleAppData(List<Object> data) {
               buf1.get(0).destID = aggrID; // send to aggregator peer
               buf1.get(0).data = data;
               List<PMData> tmp = buf1;
               buf1.clear();
               return tmp;
        }

        public List<PMData> handlePeerData(List<PMData> data) {
               buf2.addAll(data);
               if (buf2.size() < level) return null;
               else {
                       List<PMData> tmp = buf2;
                       buf2.clear();
                       return tmp;
               }
        }

        public void onLevelUpdate (int privLevel) {
               level = privLevel;
        }

        public void onPeerGroupUpdate(List<PeerInfo> peers) {
               aggrID = smallestPeerID(peers);
         }
}
```

Listing 3. Simple collaborative privacy mechanism for the hotspot sensing task, which anonymizes the application data.

It is important to stress the fact that a collaborative privacy mechanism can adapt its operation as a function of the current peer group configuration. The respective updates are handled via the `onPeerGroupUpdate()` method. In our prototype, the information that is passed to the privacy mechanism is really minimal, consisting of the identifier and current privacy level of each peer that is in rage of the local device. But it is straightforward to extend the implementation in order to include additional information about each peer.

4.4 Peer-to-peer interaction support

The peer-to-peer group formation and message passing used for the collaborative privacy mechanisms is based on Android's Wi-Fi Peer-to-Peer[4] (WiFiP2P) subsystem, which complies with the Wi-Fi Direct™ certification of the Wi-Fi Alliance. Using these APIs, devices that are close to each other can discover, identify and connect to each other without requiring an intermediate access point or going on the Internet. The first time a connection is attempted between two devices, the owner of the target device is prompted to accept (or decline) the connection, so the user is in control of the peer group formation. Once a pairing is successfully established, subsequent interactions can occur in the background, without asking for user permission.

[4] developer.android.com/guide/topics/connectivity/wifip2p.html

When the *EasyHarvest* client instantiates a collaborative privacy mechanism, it registers with WiFiP2P a corresponding service with the client's identifier, the identifier of the privacy mechanism, and the privacy level. From that point onwards, the client is automatically notified about the presence of other peers, and in turn informs the local privacy mechanisms, via the `onPeerGroupUpdate()` method, each time a compatible peer (one that runs the same privacy mechanism) is discovered or disappears. When a privacy mechanism requests data to be sent to a peer (instead of the server), the client performs the data transfer using the WiFiP2P primitives; at the destination, the client calls the `handlePeerData()` method of the local privacy mechanism component.

Chapter 5 Evaluation

We evaluate the overhead of our implementation using Android devices that run the *EasyHarvest* client software. The *EasyHarvest* server runs on a PC. Client-server communication is over a commodity Wi-Fi access point. Measurements are taken via the Dalvik Debug Monitor Server[5] and the GSam Battery Monitor[6], on an eSTAR tablet with a 1.2 GHz Allwinner A33 Quad-core ARM Cortex-A7 CPU, 512 MB RAM, 2500 mAh Li-Po battery, running Android 4.2.2. To capture the resource consumption of the privacy framework, we study the following configurations:

- **No privacy (No-Priv):** The client runs a sensing task without using a privacy mechanism for it, so the data that is produced by the task is sent directly to the server.

- **Standalone privacy (S-Priv):** The client runs a sensing task with a “null” privacy mechanism, which simply forwards the data produced by the task to the server.

- **Collaborative privacy (C-Priv):** The client runs a sensing task with a “null” privacy mechanism, where the data that is produced by the sensing task is sent to a distinguished peer, which then forwards it to the server.

The No-Priv configuration serves as a baseline for S-Priv and C-Priv. In all cases, we use a simple sensing task that periodically reads the GPS and returns a location/time data item. In the spirit of a “null” function, the privacy mechanisms used in S-Priv and C-Priv merely copy but do not inspect or process the data of the sensing task. The client is configured to send data over the network (to the server or a peer) at the same rate at which it is produced by the sensing task.

The No-Priv and S-Priv configurations are tested using a single device. For C-Priv, two devices are used, one of them acting as the data forwarder towards the server. We run our experiments for a data production/upload period of 10 seconds. This rather aggressive period is chosen to accentuate the differences – most applications would work at a much more infrequent interaction. In S-Priv, each upload involves the transfer of a single data item, whereas in C-Priv the forwarder sends to the server two data items at a time (the one produced locally, and the one produced by the other peer). In all configurations, the client checks the server for task status and parameter updates, at the same period. Every experiment runs for 1 hour, with the screen and Wi-Fi of the tablet turned on all the time.

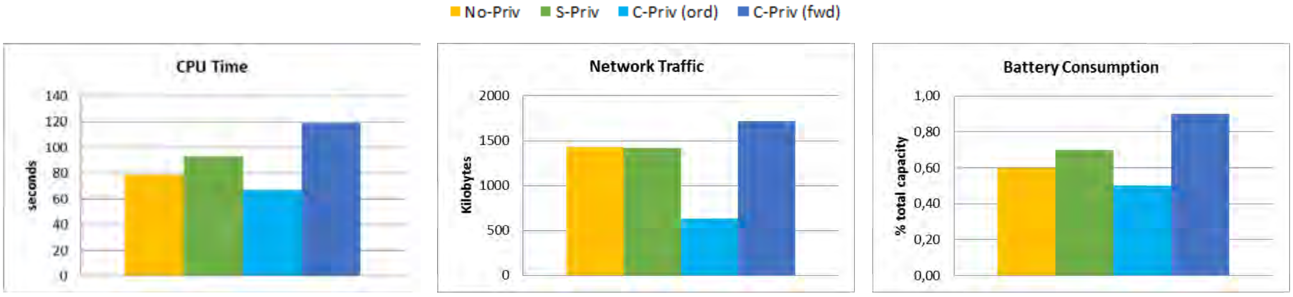

Figure 5. CPU, network and battery usage, at a data production and upload period of 10 seconds, for 1 hour of continuous operation.

The results are shown in Figure 5. S-Priv produces the same network traffic as No-Priv, but has an increased CPU load, attributed to the data copying performed by the privacy mechanism. This in turn leads to a higher battery consumption. In C-Priv, the overhead for the ordinary peer is actually lower than that of S-Priv, even though the data produced by the sensing task is the same in both cases. This is because in S-Priv this data is sent to the server via heavyweight REST-based interactions, while in C-Priv data is sent to the forwarder peer via WiFiP2P in more efficient way, leading to less networking and CPU activity. However, the overhead for the forwarder in C-Priv is significantly higher than S-Priv, as the forwarder receives the data of the ordinary peer over WiFiP2P and uploads it to the server, along with the data that is produced locally. This leads to an increased network activity, stronger CPU usage, and

[5] developer.android.com/tools/debugging/ddms.html
[6] play.google.com/store/apps/details?id=com.gsamlabs.bbm

higher battery consumption. Memory consumption remained stable, at 3.4-3.5 MB, with negligible differences and variations between the different configurations. We conducted additional experiments for larger data production and upload periods. The results follow similar trends, while the absolute resource consumption drops in all cases as the period increases.

Chapter 6 Related Work & Discussion

There has been some related work on privacy and sensor-data access-control frameworks for smartphone applications. In the sequel, we provide an overview of indicative approaches and compare them with our privacy mechanism framework.

BlurSense [2] is a dynamic fine-graned access control mechanism that provides secure and customizable access to the sensors on mobile devices, and allows the definition and installation of privacy filters. BlurSense runs in an isolated sandbox and employs Sensorium, a unified sensor interface, to access sensor data. However, the approach of BlurSense requires specific hooks to be developed so that sensor data requests from generic apps to be sent through BlurSense with XMLRPC, as opposed to our simple sensor data interface. Privacy filters are supposed to be developed by security vendors in BlurSense. These filters do not take into account the context of the user device, while collaborative privacy among different smartphones is not considered.

CRePE [5] allows the definition of context-related policies either by users or by trusted third parties. Third parties can set on a device context independent policies that apply for any moment or dynamic rules adaptive to context alternations. The main novelty of CRePE is the introduction of context awareness into the access control mechanism. However, its focus is quite different from ours: in effect, the aim is to change Android's permissions dynamically according to context.

Aurasium [17] is a policy enforcement framework for Android applications that automatically repackages arbitrary applications to attach user-level sandboxing and policy enforcement code. Thus, Aurasium monitors applications' behavior for security and privacy violations (e.g. sensitive information disclosure, SMS covertly charging, malicious URL access, etc.) and is able to detect and prevent cases of privilege escalation attacks. MPdroid [11] is a security framework for Android which supports the enforcing of multiple security policies. It allows users to define their own security policy and provides fine-grained access control to (untrusted) applications. Both Aurasium and MPdroid are of the on-off type (which is what most access control software does).

Secure Application INTeraction (Saint) [14] is an install and run-time application management system for Android. It allows to enforce policies that enable applications to define which other applications can access their interfaces, how those interfaces are used, and select at runtime which of the application's interfaces to use. Unlike our work, Saint focuses on protecting an application from other applications running on the same device, rather than on transforming the data that is uploaded to the Internet by mobile crowdsensing scenario in order to protect user privacy.

AnonySense [6] is a system for enabling opportunistic sensing applications, while hiding a user's location among *k* users, on average. In addition, attribute values that are reported by users may be either generalized (i.e. made less specific) or suppressed altogether so as to make each user's report identical to *l* other reports. However, these privacy objectives, as well as the values of *k* and *l*, are fixed and cannot be chosen by users. As such, AnonySense is not a privacy framework, but rather one (of many other possible) privacy mechanisms that can be accommodated via our framework. PEPSI [8] takes an Identity-Based Encryption approach to provide unlinkability for the mobile nodes and for the queries in a participatory sensing context with minimal trust to third parties. However, the sensor-related data itself does not undergo any privacy-enhancing transformation.

The PRISM platform [7] controls access to the sensors of the smartphone, either in a coarse-grained manner or by means of application-specific energy-usage and bandwidth-usage limits. To prevent sensor data accumulation, PRISM employs "forced amnesia", periodically clearing the application state. TaintDroid [9], similarly to PRISM, tracks sensor data access and usage by various smartphone applications, in order to increase user awareness on privacy leakage and sensor data misuse.

SemaDroid [18] is a framework for controlling access and usage of sensors through hooks in Android for intercepting sensor data requests from the various applications. SemaDroid supports user-defined privacy policies for the various sensor-application pairs. It mocks the generation of sensor data, when access to this data is restricted, in order this restriction to go unnoticed by the application. However, SemaDroid does not consider privacy-enhancing transformations of sensor data or collaborative privacy schemes. Also, mock-up data may harm application utility.

Finally, ipShield [3] tracks the usage of every sensor employed by an app and it performs a privacy-risk assessment and presents this information to the user. ipShield recommends possible privacy actions based on user preferences, and allows users to define context-aware fine-grained privacy rules, through which one can suppress, constant, perturb and playback sensor data.

The main advantage of our approach compared to on-off access control mechanisms is that it becomes possible to design and implement flexible privacy mechanisms that explicitly consider the utility of the data items that are produced by the mobile part of the crowdsensing application, and apply the transformations that are required to protect specific user privacy aspects. Moreover, low-level on-off mechanisms cannot work in a collaborative way, as this needs to be done at a higher level, taking into account the type of data that is produced by the mobile application as well as its semantics. Note that in our framework, if desired, it is still possible to emulate an on-off approach, using a privacy mechanism that completely blocks a given type of sensor data (instead of trying to transform it). Last but not least, the idea of dynamically pluggable data filtering/transformation logic can be applied at the lower-level sensor interface of the runtime / operating system as well.

One potential issue with the proposed framework is that the application might attempt to bypass the privacy mechanism, and covertly send privacy-sensitive information via a hidden application-internal encoding, in the value and/or time domain. But the framework also provides the means for introducing effective countermeasures. As mentioned, the values of application data can be formally defined – and constrained – as part of the data model that can be safely checked at compile and run time. Moreover, value-based encodings are cancelled by privacy mechanisms that distort the data values, while time-based encodings are invalidated if the data is buffered locally for some random amount of time before sending it to the server. While hidden signals cannot be prevented if the application’s utility model does not allow such interventions, it is also doubtful that such an application would work in an acceptable way on top of low-level privacy mechanisms, which completely block one or more data feeds or silently distort data under the hood before this is even read by the application.

Chapter 7 Conclusion & Outlook

We have presented a conceptual framework for separating privacy mechanisms from the sensing part of crowdsensing applications that run on personal devices, and developing such mechanisms as pluggable software components that can be dynamically downloaded and bound to the application at runtime. Our design achieves more flexibility compared to current low-level access control and data filtering approaches, which can unintentionally harm application utility and cannot support collaborative privacy schemes. We have also discussed a proof-of-concept implementation of the privacy framework for an existing crowdsensing system targeting Android devices, and have shown that the respective overhead is small for standalone mechanisms but can grow quite significantly for collaborative schemes where all the data that is produced by the application is aggregated on a single device.

An interesting research direction, from a data engineering perspective, would be to come up with suitable standards for application-generated data types and the corresponding utility functions for typical privacy-preserving transformations that can be applied to such data. Based on such standardized data models, one could engineer more generic privacy mechanisms, which can be re-used for different applications, adapting their operation accordingly.

Our prototype could also be extended to provide more advanced functionality. For instance, the client runtime could adjust the level of the current privacy mechanism or even switch to another mechanism that will lead to better privacy, based on contextual information, such as the user's location or activity, and the presence of other peers. Such an adaptive operation could be driven by explicit as well as learned user preferences. Further, historical evidence on prior interaction and trust relationships among users, e.g., via some online social network, could be exploited for defining more reliable/trusted peer groups. With rather surgical modifications on the server, it would also be possible for the client to engage known/trusted peers even if these are in remote locations.

References

[1] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. *17th ACM Conference on Computer and Communications Security*, 2010.

[2] J. Cappos, Wang Lai, R. Weiss, Yang Yi, Zhuang Yanyan. BlurSense: Dynamic fine-grained access control for smartphone privacy. *IEEE Sensors Applications Symposium (SAS)*, 2014.

[3] S. Chakraborty, C. Shen, K.R. Raghavan, Y. Shoukry, M. Millar, M.Srivastava. ipShield: A Framework For Enforcing Context-Aware Privacy, *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[4] D. Christin, A. Reinhardt, S.S. Kanhere, M. Hollick. A survey on privacy in mobile participatory sensing applications, *Journal of Systems and Software*, (84)11, Elsevier, November 2011.

[5] M. Conti, V. Nguyen, B. Crispo. CRePE: Context-related policy enforcement for Android. *Information Security*, Lecture Notes in Computer Science, Vol. 6531, Springer, 2011.

[6] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, N.Triandopoulos. Anonysense: privacy-aware people-centric sensing. *6th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2008.

[7] T. Das, P. Mohan, V.N. Padmanabhan, R. Ramjee, A. Sharma. PRISM: platform for remote sensing using smartphones. *8th International Conference on Mobile Systems, Applications, and Services (MobiSys*), 2010.

[8] E. De Cristofaro, C. Soriente. PEPSI---privacy-enhanced participatory sensing infrastructure. 4th *ACM Conference on Wireless Network Security (WiSec)*, 2011.

[9] W. Enck, P. Gilbert, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel, A.N.Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[10] Gartner’s press release on smartphone sales for 2014, electronically available at www.gartner.com/newsroom/id/2996817.

[11] T. Guo, Z. Puhan, L. Hongliang, S. Shuai. Enforcing Multiple Security Policies for Android System. *2nd International Symposium on Computer, Communication, Control and Automation*, 2013.

[12] M. Katsomallos, S. Lalis. EasyHarvest: Supporting the Deployment and Management of Sensing Applications on Smartphones, *1st International Workshop on Crowdsensing Methods, Techniques and Applications*, in conjunction with *12th IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2014.

[13] G. Kazdaridis, D. Stavropoulos, V. Maglogiannis, T. Korakis, S. Lalis, L. Tassiulas. NITOS BikesNet: Enabling Mobile Sensing Experiments through the OMF Framework in a City-wide Environment, *15th IEEE International Conference on Mobile Data Management (MDM)*, 2014.

[14] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel. Semantically rich application-centric security in Android. *Security and Communication Networks*, 5(6), June 2012.

[15] A. Pantelopoulos and N.G. Bourbakis, A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis, IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews, 10(1), January 2010.

[16] D. P. Rose, M. Ratterman, D. K. Griffin, L. Hou, N. Kelley-Loughnane, R.R. Naik, J.A. Hagen, I. Papautsky and J. Heikenfeld, Adhesive RFID Sensor Patch for Monitoring of Sweat Electrolytes, IEEE Transactions on Biomedical Engineering, 62(6), June 2015.

[17] R. Xu, H. Saïdi, R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications, *USENIX Security Symposium*, 2012.

[18] Z. Xu, S. Zhu. SemaDroid: A Privacy-Aware Sensor Management Framework for Smartphones. *5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.