An Open Framework for Flexible Plug-in Privacy Mechanisms in Crowdsensing Applications

Manos Katsomallos*†, Spyros Lalis†, Thanasis Papaioannou†, and George Theodorakopoulos‡
emmanouil.katsomallos@u-cergy.fr, {lalis,atpapaioannou}@uth.gr, TheodorakopoulosG@cardiff.ac.uk
*ETIS/ENSEA - University of Cergy-Pontoise - CNRS, Cergy-Pontoise, France
†University of Thessaly & CERTH, Volos, Greece
‡Cardiff University, Cardiff, UK

***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 that decouples the privacy mechanism from the application logic so that it can be developed by another, perhaps more trusted party, and which allows the dynamic binding of different privacy mechanisms to the same application running on the user's mobile device. We describe a proof-of-concept implementation of the proposed framework for Android, where privacy mechanisms are independently developed as separate plug-in components. Based on a simple but powerful API, it is possible to implement a wide range of standard privacy approaches, including collaborative schemes that involve data exchanges among multiple personal devices.**

I. Introduction

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 battery lifetime, simplify the installation and management of sensing applications on the smartphone, and address various privacy and trust concerns.

The privacy issue is of key importance in order for people to agree/volunteer to provide data via their mobile personal devices. As a result it has 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. Moreover, these implementations are tightly-coupled with the application logic, making them hard to be inspected, replaced, let alone reused in other applications. Also, it is doubtful that one size will fit all. Namely some users may have different privacy concerns than others, and the privacy needs of the same user may be context-dependent.

Motivated by the above observations, we propose an open framework that enables 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 compatible sensing applications running on the device. We also describe a prototype implementation of the proposed framework for *EasyHarvest* [10], a crowdsensing system for mobile devices. Note that our contribution is the software framework that enables a flexible deployment of different privacy mechanisms, not the privacy mechanisms themselves.

The main features of our framework are briefly as follows: (i) Privacy mechanisms are developed using a simple yet powerful API, which also supports the implementation of collaborative schemes. (ii) Privacy mechanisms are developed as plug-in components, which can be dynamically loaded and linked to compatible sensing applications running on personal devices. (iii) Users can review the privacy mechanisms that are available, and pick the ones that seem most suitable for their needs. (iv) Users can set the desired level of privacy for different spatio-temporal regions, letting the system adjust the operation of the privacy mechanisms accordingly.

The rest of the paper is structured as follows. Section II introduces the proposed concept. Section III gives an overview of the *EasyHarvest* system. Section IV describes a proof-of-concept implementation of the proposed privacy framework for it. Section V gives an overview of related work. Finally, Section VI concludes the paper and identifies directions for future work.

II. 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 these threats [3], e.g., 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 typically come with hardwired privacy protection support, if any. Even if the code is open for inspection, the

average user 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 contributing 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.

Locality: Preserve privacy locally, on the personal device or among a group of trusted peer 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 they can intercept and process/filter privacy-sensitive data without blocking low-level data feeds. 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 preserving application utility.

Hereinafter, 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 that will run on the personal device. The data model defines the data items and values that are produced by the sensing component 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 the data model, and provide suitable privacy mechanisms, 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, users who wish to contribute to a crowdsensing effort pick and install the respective sensing component along with one or more compatible privacy mechanism components.

On the personal device, a suitable runtime system is needed in order to support the dynamic installation, binding and execution of these software components. Figure 2 shows

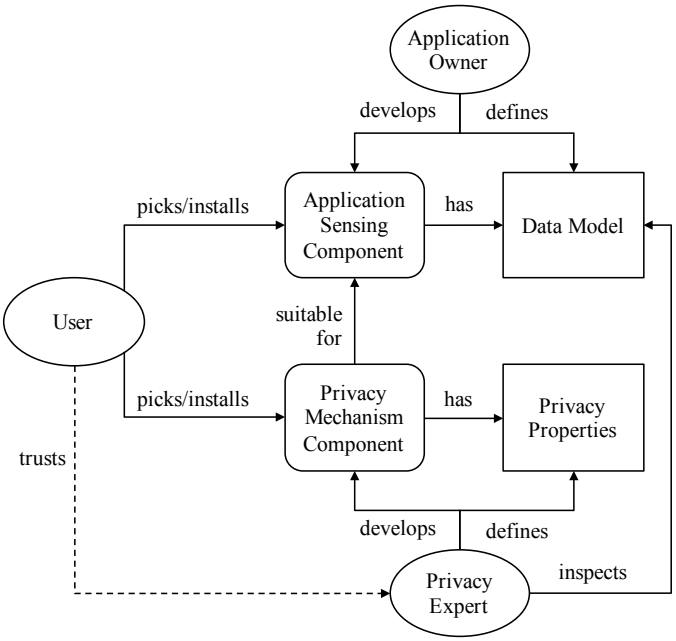

Fig. 1: Structured introduction of privacy mechanisms for crowdsensing applications: stakeholders and software artifacts.

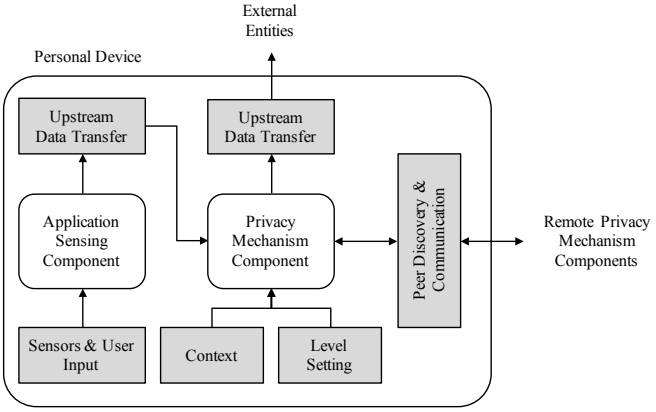

Fig. 2: High-level runtime architecture on the personal device. Grey boxes stand for the system interfaces/hooks.

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 in a context-driven way.

Note that there is no attempt to control the type and/or amount of data that is locally accessed by the sensing component of the application. The information gathered on the personal device is considered harmless, as long as it does not propagate to the Internet—and presicel this upstream flow goes through the privacy mechanism, under the control of the runtime system. Of course, the application should declare its data model truthfully. But doing so is in its own interest. False declarations concerning the value ranges and utility of the data will result to the application being associated with an inappropriate privacy mechanism that might easily destroy application data. Also, assuming strong typing support, 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.

III. The EasyHarvest System

The *EasyHarvest* system [10] was designed to simplify the development, managed deployment and controlled execution of sensing tasks on personal devices. Each sensing task is an application-specific mobile 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 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.

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/deactivates the sensing task depending on the spatio-temporal context of the device. Post-processing of the data collected on the server, which can be retrieved data via a suitable machine interface, is left for external, application-specific subsystems.

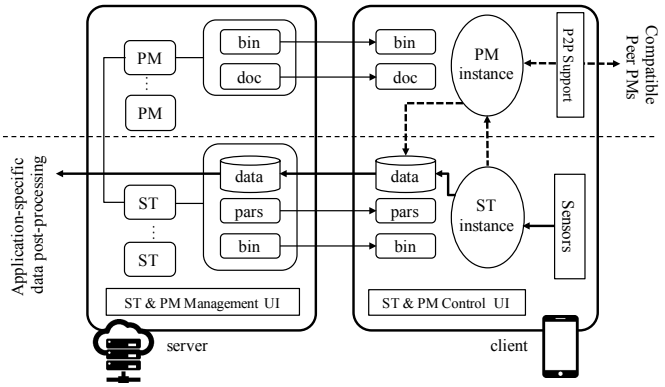

Fig. 3: High-level architecture of *EasyHarvest*. The client retrieves tasks (STs) from the server, and runs them on the personal device. The extensions for the plug-in privacy mechanisms (PMs) are shown above the dashed line.

TABLE I: Sensing Task Interface

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.

The current *EasyHarvest* client prototype is designed for Android devices. Sensing tasks have the form of Java classes for the Dalvik environment , and implement a predefined interface Table I though which they interact with and are controlled by the client runtime.

Sensing tasks access the sensors (and possibly other data sources) of the smartphone via the native Android API. 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 task binary. At runtime, the *EasyHarvest* client performs type-checks to verify that the task indeed produces the expected type of data.

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 system 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).

IV. 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 discuss in detail the implementation.

A. Privacy Mechanism Registration & 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). The code of the mechanism is open-source and available for inspection by community members, in particular other privacy experts.

Note that the description of the privacy mechanism is written by the privacy expert so that it is comprehensible even by novice users. The privacy mechanism may include, as part of its metadata, a user-friendly explanation of the expected privacy results, e.g., "your location will get distorted by x meters" or "you will be hidden among y people". Moreover, the privacy expert provides the expected utility deterioration per privacy level for each application sensing task. This could be useful for choosing among privacy mechanisms that provide equivalent privacy.

As part of its periodic interaction with the server, the *EasyHarvest* client queries for 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, reads the description and privacy features and selects the one to employ for the application. In turn, the client downloads the privacy mechanism on the device, and instantiates/binds it to the sensing task. At runtime, the user sets the privacy level of the mechanism.

B. Interface of Privacy Mechanism Components

Privacy mechanisms are implemented as Java classes for the Dalvik environment. By convention, a privacy mechanism shall provide a pre-defined interface Table II. Note that only a subset of this interface is mandatory (underlined).

The data interface of privacy mechanisms refers to abstract data objects, just as this is the case for application sensing tasks. However, the concrete data items that will be fed into the 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.

C. 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. Note that the privacy mechanism can record arbitrary state information via the `onStop()` method, and then retrieve it again via the `onStart()` method. This way, it is possible for future executions of the mechanism to exploit information/experience that was produced in the past.

Standalone privacy mechanisms do not provide the `handlePeerData()` and `onPeerGroupUpdate()` methods. When the *EasyHarvest* client detects that these methods are missing, it sets 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 that the data should be sent directly to the server.

TABLE II: Privacy Mechanism Interface

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> handlePeerData( 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.

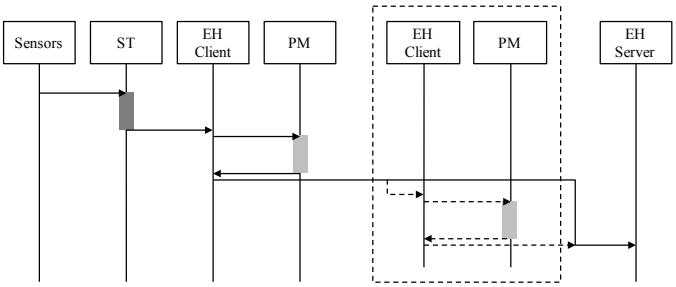

Fig. 4: Data flow for standalone (solid lines) and collaborative privacy mechanisms (dashed lines).

Collaborative privacy mechanisms have to implement the full interface. Method `handleAppData()` is used to forward locally generated application data to other peers for further processing and aggregation. Data arriving from other peers is processed via the `handlePeerData()` method. Note that it is 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 peer, which can further process the data before sending it to the server, as illustrated in Figure 4 (dashed lines).

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 passed to the privacy mechanism is minimal, consisting of the identifier and current privacy level of each peer that is in rage of the local device. It is straightforward to extend the current implementation in order to include additional information about each peer.

D. Privacy Settings & Dynamic Privacy Adjustment

The *EasyHarvest* framework makes it possible for the user to set a desired privacy level that applies to the installed privacy mechanism and define so-called privacy regions (in time and space) where all sensing tasks are suspended. Furthermore, it offers the option for a more advanced, adaptive behavior in terms of privacy, depending on the user's privacy settings and context. This way, the user is relieved from repeatedly adjusting the current privacy level, and manually switching between different privacy mechanisms.

This adaptive operation is introduced in conjunction with the privacy regions that can be freely defined by the user. More specifically, the user may associate each region with a different privacy level: "low", "medium", "high" and "block". Regions that are not characterized, are considered "free". The *EasyHarvest* client tracks the user's spatiotemporal context and adapts the operation of the privacy mechanisms accordingly.

While in a "free" region, the privacy mechanism (if any) is deactivated allowing the sensing application task to report data without any privacy checks. When in a "blocked" region, the mechanism and sensing application are deactivated (all sensing activity is halted). In all other cases, if there is a privacy mechanism that can be used for the sensing application or a mechanism is already being used for this sensing task, the *EasyHarvest* client activates it and sets its privacy level via `onLevelUpdate()` according to the user's preference ("low", "medium", "high"). If there is no compatible privacy mechanism, the sensing task is deactivated.

Finally, standalone privacy mechanisms might be considered safer than collaborative ones. To support such a preference, the *EasyHarvest* client can be configured to automatically switch from a collaborative to a standalone privacy mechanism (provided that such a mechanism exists for the sensing task) when the user enters a "high" privacy region.

E. 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 (Wi-Fi P2P) subsystem. Using this API, devices that are close to each other can discover, identify and connect 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.

When the *EasyHarvest* client instantiates a collaborative privacy mechanism, it registers with Wi-Fi P2P 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 notified about the presence of other peers, and in turn informs the local privacy mechanisms, via the `onPeerGroupUpdate()` method, each time a peer that runs the same privacy mechanism is discovered or disappears. When a privacy mechanism requests data to be sent to a peer, the client performs the data transfer using the Wi-Fi P2P primitives; at the destination, the *EasyHarvest* client delivers this data by calling the `handlePeerData()` method of the local privacy mechanism component.

V. Related Work

BlurSense [1] is a dynamic fine-grained access control mechanism that provides secure and customizable access to the sensors on mobile devices, and allows the definition and installation of privacy filters (these are supposed to be developed by security vendors). BlurSense runs in an isolated sandbox and employs Sensorium, a unified sensor interface, to access sensor data. This requires specific hooks to be developed in order for the application's sensor data requests to actually go through BlurSense. Also, the filters of BlurSense do not take into account the context of the personal device (location or previous history of data emissions), and cannot work in a collaborative way (between smartphones).

CRePE [4] allows the definition of flexible access control policies. These can be fixed and apply at all times, or adapt in a dynamic way based on context-sensitive rules. CRePE is designed to change Android's permissions according to context, but the same principles could be used in our system to switch between different privacy mechanisms.

Aurasium [11] is a policy enforcement framework for Android applications that automatically repackages applications to attach user-level sandboxing and policy enforcement code. It monitors applications' behavior for security and privacy violations (sensitive information disclosure, SMS covertly charging, malicious URL access) and can detect/prevent cases of privilege escalation attacks. MPdroid [9] is a security framework for Android which supports multiple security policies. It lets users define their own policy and provides fine-grained access control to (untrusted) applications. Both Aurasium and MPdroid follow an "on-off" access model (as most systems).

AnonySense [5] supports opportunistic sensing applications, while hiding a user's location among those of several users. In addition, attribute values reported by users may be either generalized or suppressed to make each user's report identical to a number of other reports. However, these privacy objectives, as well as the anonymization parameters are fixed and cannot be chosen by users. As such, AnonySense is not a privacy framework, but rather one (of many possible) privacy mechanisms that can be accommodated via our framework. PEPSI [7] takes an Identity-Based Encryption approach to provide unlinkability for the mobile nodes and the queries in a participatory sensing context with minimal trust in third parties. However, sensor-related data does not undergo any privacy-enhancing transformation.

The PRISM platform [6] 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 [8], similarly to PRISM, tracks sensor data access by various smartphone applications, to raise user awareness on privacy leakage and sensor data misuse.

SemaDroid [12] is a framework for controlling sensor access/usage through hooks in Android that intercept sensor data requests from various applications. SemaDroid supports user-defined privacy policies for the various sensor-application pairs, and transparently provides mocked data to the application when access to the respective sensors is restricted. 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 [2] tracks the usage of every sensor employed by an app and performs a privacy-risk assessment presenting this information to the user. ipShield recommends possible privacy actions based on user preferences, and enables users to define context-aware fine-grained privacy rules, that can suppress, perturb and playback sensor data.

VI. Conclusion & Outlook

We have proposed a concept for separating privacy mechanisms from the sensing part of crowdsensing applications that run on personal devices, and developing such mechanisms as plug-ins that can be flexibly bound to the application. We also presented a proof-of-concept implementation as an extensions of a crowdsensing system for Android devices.

The proposed approach allows to design and implement flexible privacy mechanisms, which go beyond the "on-off" model of current access control mechanisms, and to consider the utility of the data items produced by the mobile part of the crowdsensing application. It also allows to implement collaborative privacy mechanisms, which is impossible with low-level "on-off" mechanisms. Of course, it is still possible to emulate an "on-off" approach, is desired, using a privacy mechanism that simply drops a given type of data.

An interesting research direction would be to define standards for application-level data types and corresponding utility functions for privacy-preserving transformations that can be applied to such data. Based on such models, one could engineer generic privacy mechanisms, which can be re-used for different applications. Several extensions can also be made in our prototype. For instance, more intelligent strategies could be employed for the dynamic selection/switching of privacy mechanisms, based on the utility each mechanism can achieve for the same privacy level setting. Further, historical evidence on prior interaction/trust among users (e.g., via social networks) could be exploited to define more reliable peer groups for the collaborative privacy mechanisms; with surgical modifications on the server, the client could engage known/trusted peers even if these are in remote locations. Last but not least, privacy mechanisms could let the user "callibrate", in a personalized manner, key algorithmic parameters with respect to the generic privacy levels (low, medium, high) assigned to the different regions, rather than using some hardwired internal values for all users.

References

[1] Justin Cappos, Lai Wang, Rebecca Weiss, Yi Yang, and Yanyan Zhuang. Blursense: Dynamic fine-grained access control for smartphone privacy. In *Proc. IEEE Sensors Applications Symposium*, pages 329–332, 2014.

[2] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. ipshield: a framework for enforcing context-aware privacy. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation*, pages 143–156, 2014.

[3] Delphine Christin. Privacy in mobile participatory sensing: current trends and future challenges. *Journal of Systems and Software*, 2015.

[4] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Information Security*, pages 331–345. Springer, 2011.

[5] Cory Cornelius, Apu Kapadia, David Kotz, Dan Peebles, Minho Shin, and Nikos Triandopoulos. Anonysense: privacy-aware people-centric sensing. In *Proc. 6th International Conference on Mobile systems, Applications, and Services*, pages 211–224, 2008.

[6] Tathagata Das, Prashanth Mohan, Venkata N Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. Prism: platform for remote sensing using smartphones. In *Proc. 8th International Conference on Mobile systems, Applications, and Services*, pages 63–76, 2010.

[7] Emiliano De Cristofaro and Claudio Soriente. Short paper: Pepsi—privacy-enhanced participatory sensing infrastructure. In *Proc. 4th ACM Conference on Wireless Network Security*, pages 23–28, 2011.

[8] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2):5, 2014.

[9] Tao Guo, Puhan Zhang, Hongliang Liang, and Shuai Shao. Enforcing multiple security policies for android system. In *Proc. 2nd International Symposium on Computer, Communication, Control and Automation*, 2013.

[10] Manos Katsomallos and Spyros Lalis. EasyHarvest: Supporting the deployment and management of sensing applications on smartphones. In *Proc.IEEE International Conference on Pervasive Computing and Communications (Workshops)*, pages 80–85, 2014.

[11] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Proc. USENIX Security Symposium*, pages 539–552, 2012.

[12] Zhi Xu and Sencun Zhu. Semadroid: A privacy-aware sensor management framework for smartphones. In *Proc. 5th ACM Conference on Data and Application Security and Privacy*, pages 61–72, 2015.