VIVO: a Secure, Privacy-Preserving, and Real-Time Crowd-Sensing Framework for the Internet of Things

Luca Luceri[a,b], Felipe Cardoso[a], Michela Papandrea[a], Silvia Giordano[a], Julia Buwaya[c], Stéphane Kundig[c], Constantinos Marios Angelopoulos[c,e], José Rolim[c], Zhongliang Zhao[b], Jose Luis Carrera[b], Torsten Braun[b], Aristide C. Y. Tossou[d], Christos Dimitrakakis[d], Aikaterini Mitrokotsa[d]

[a] *SUPSI, Switzerland*
[b] *University of Bern, Switzerland*
[c] *University of Geneva, Switzerland*
[d] *Chalmers University, Sweden*
[e] *Bournemouth University, UK*

Abstract

Smartphones are a key enabling technology in the Internet of Things (IoT) for gathering crowd-sensed data. However, collecting crowd-sensed data for research is not simple. Issues related to device heterogeneity, security, and privacy have prevented the rise of crowd-sensing platforms for scientific data collection. For this reason, we implemented VIVO, an open framework for gathering crowd-sensed Big Data for IoT services, where security and privacy are managed within the framework. VIVO introduces the *enrolled crowd-sensing* model, which allows the deployment of multiple simultaneous experiments on the mobile phones of volunteers. The collected data can be accessed both at the end of the experiment, as in traditional testbeds, as well as in real-time, as required by many Big Data applications. We present here the VIVO architecture, highlighting its advantages over existing solutions, and four relevant real-world applications running on top of VIVO.

Keywords: Mobile Crowd-Sensing, Internet of Things, Big Data

This work is supported by the Swiss National Science Foundation via the SwissSenseSynergy project, grant number 154458.

1. Introduction

Smartphones have completely revolutionized our life, work, and free time with the tremendous growth of novel resources and services. Smartphones are truly portable, personal, and highly connected devices: as such, they are a key enabling technology in the Internet of Things (IoT), where people produce crowd-sensed data. However, collecting such data for research is not simple; contributors need to be actively enrolled in a campaign, and thus issues related to device heterogeneity, security, and privacy need to be considered. Such difficulties have prevented the rise of crowd-sensing platforms for scientific data gathering. Similarly, the use of previously gathered crowd-sensed data is hard. Rarely such data are appropriate for the intended study, and thus require further assumptions and filtering. Thus, many potential crowd-sensing services have not (yet) been recognized due to the lack of adequate testing data.

To this aim, we implemented VIVO, an open framework for crowd-sensed Big Data gathering, where security and privacy are managed within the framework at the client side. VIVO allows to test and validate IoT services that use social, physical, and environmental information. The collected data can be accessed both at the end of the experiment, as in traditional testbeds, and in real-time, as required by many big data applications. Yet, VIVO differs from traditional testbeds as testing experiments can be scheduled and run in *real-time* on the mobile phones of volunteers. Here, we present the following contributions:

- the introduction of the *enrolled crowd-sensing* model that allows the deployment of several experiments simultaneously, as opposed to the traditional usage of crowd-sensing for a single experiment;
- a paradigm-shift from (*i*) taking care of the whole experiment cycle, i.e., from the experiment design up to the data provision, to (*ii*) managing only the experiment application, with built-in security and privacy capabilities;
- the VIVO architecture definition and implementation, its performance evaluation and, as an example, four relevant real-world applications.

2. Background and Motivation

Mobile Crowd-Sensing (MCS) is an emerging paradigm based on the sensing capabilities of mobile devices [1]. MCS lies at the intersection between the IoT and the volunteer/crowd-based scheme [2]. In particular, MCS extends IoT services relying on data collected from a large number of individuals' portable sensing devices, such as smartphones. Potential MCS applications span a wide spectrum in terms of application domain [3], ranging from environmental monitoring [4, 5, 6], traffic estimation [7, 8, 9], and place categorization [10] to smart cities [11, 12, 13] or buildings [14], and social trend detection [15, 16]. Though these applications were established to pursue specific purposes, efforts have also been made towards formally characterizing the operation of MCS systems in an application-agnostic way. These approaches offer more flexibility by supporting a variety of experiments in different settings, ranging from participatory to opportunistic sensing, depending on the user involvement in the data collection scheme [17, 18, 19]. In [20, 21], we identified basic design issues of MCS systems and investigated some characteristic challenges. In [22], authors recognize the opportunity of fusing information from populations of privately-held sensors as well as the corresponding limitations due to privacy issues.

Inspired by this fruitful ensemble of works, we introduce a novel crowd-sensing testbed, referred to as VIVO. The key point of our proposed solution consists in allowing an easy development and deployment of experimental software on mobile devices. More precisely, similarly to PhoneLab [23] and SmartLab [24], VIVO *experiment developers* (i.e., application developers who need to collect data) can dynamically deploy their own application on each VIVO volunteer device. However, while PhoneLab [23] requires volunteers to run a modified version of the Android OS on their mobile phone, thus limiting the set of potential participants, VIVO experimental applications run on standard Android versions, without any extra-hardware requirements and pre-deployment testing. SmartLab [24] is an architecture for managing a cluster of real and virtual devices. Users can install executables on devices, capture their screen,

Table 1: Testbeds Comparison

	LiveLabs	NetSense	PhoneLab	IoTLab	**VIVO**
standard smartphone OS	✓	✓	✗	✓	✓
simultaneous experiments	✗	✗	✓	✓	✓
open range of applications	✗	✗	✓	✗	✓
real-time data collection	✓	✗	✗	✓	✓
embedded security	✓	✓	✗	✗	✓
privacy-preserving	✓	✓	✗	✓	✓
fixed and mobile sensors	✗	✗	✗	✓	✓

and issue UNIX shell commands. While Smartlab is targeted towards scenarios requiring low-level control over smartphones, e.g., deployment and debugging, VIVO is a framework focused on the gathering of crowd-sensed data.

Recent similar efforts are LiveLabs [25], NetSense [26], and IoT Lab [27]. Livelabs [25] is a mobile testbed that continuously collects sensor data from participant personal devices in four public spaces in Singapore. The goal of this data collection is to analytically extract context information to trigger consumer trials provided by retailers or service providers. NetSense [24] aims to understand the impact of the digital world (mobile communications and online social networks) on social relationships by collecting sensor data from instrumented smartphones distributed to hundreds of students at the University of Notre Dame. IoT Lab [27] has been developed with the purpose of researching the potential of crowd-sensing as an extension to the traditional IoT infrastructure. Through a smartphone application, the crowd was allowed to participate in experiments by contributing with sensory data and knowledge.

Unlike these previous efforts, where a single static application is installed on each smartphone to constantly save data collected from sensors, VIVO allows the deployment of multiple simultaneous experiments introducing an enrolled crowd-sensing model. In such a model, developers are not limited to a fixed set of experiments but they can build their own application without any constraint, in a more agnostic and generic way. Table 1 compares VIVO with existing solutions in the literature. Differently from other approaches, the data collected through

VIVO can be accessed both at the end of the experiment, as in traditional testbeds, as well as in real-time, as needed by several Big Data applications. This enables a broad range of applications that require low latency communication, e.g., navigation, monitoring, and recommendation.

One of the key features of VIVO concerns the security and privacy of volunteer data. As we leverage private smartphones, it becomes crucial to ensure that any deployed applications do not compromise the private data of the users and the regular behavior of their private applications. To deal with this issue, we manage security and privacy within the framework, at the client side. We provide an API with all the methods necessary to secure and privatize the collected data before they leave the smartphone. Clearly, we cannot prevent malicious behaviors, but these are legally prosecutable as a contract violation.

Moreover, VIVO is a human- and sensor-based testbed. It integrates two components: a crowd-sensing scheme composed of mobile devices (volunteer smartphones), and Syndesi [28], an IoT framework for smart buildings, which includes multiple fixed sensors. This integration empowers the seamless combination of resources coming from different sources, which (*i*) allows to study the interaction between human beings and the surroundings, analyzing their behavior with varying environmental conditions, and (*ii*) enables a big number of experiments, where users and the sensor-based infrastructure rely on each other, e.g., indoor navigation and smart actuations in the environment [29].

Finally, VIVO allows a paradigm shift from (*i*) taking care of the whole experiment cycle, i.e., from the experiment design up to the data provision, to (*ii*) managing only the experiment application, with built-in security and privacy capabilities. In fact, it provides to experiment developers a compact unified framework to collect data, from the architecture (e.g., server, data management, and security) to the mobile sensing nodes, i.e., volunteer smartphones.

Volunteer recruitment is a typical issue in crowd-sensing platforms. Thus, crowd incentives, as well as ensured Quality of Information (QoI) of crowd-sensed data, are considerably important aspects for the success of MCS applications [30, 31, 32]. To reward volunteer involvement in VIVO experiments

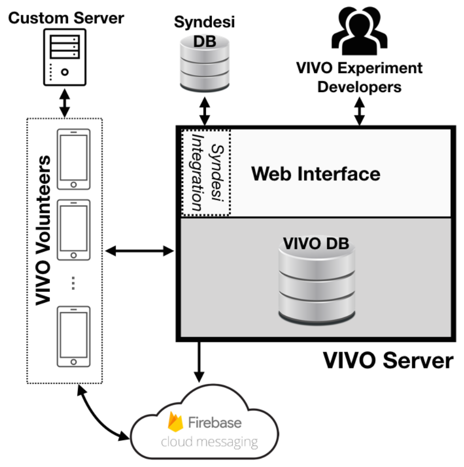


Figure 1: VIVO Architecture

we considered two strategies. First, we launched the context "Volunteer of the Year", where each participant is encouraged to participate in the largest number of experiments to win a prize. As a second step, we are working on a reward-based mechanism that allows the experiment developer to advertise prizes to the volunteers according to their involvement in the experiment.

The additional provisioning of trust and privacy along with its capability of supporting heterogeneous data (also in real-time) makes VIVO suitable for a range of diverse experiments, e.g., predicting human behavior [33], monitoring environmental conditions to examine their relation with user actions [28], or performing navigation in indoor environments [34, 35], where GPS is not usable.

3. Architectural Overview

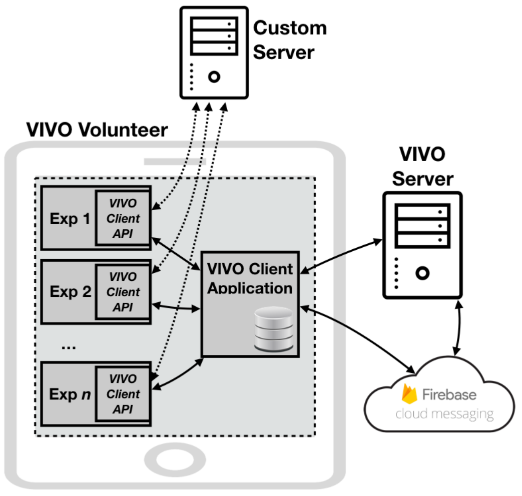
VIVO architecture is displayed in Fig. 1. At the top level, we see the VIVO experiment developers, i.e., individuals (e.g., researchers) that employ VIVO to run an experiment for collecting a dataset or testing an application. VIVO experiment developers (from now on simply referred to as developers) constitute the target group for whom VIVO has been conceived. They exploit VIVO data storage and data collection capabilities as well as VIVO volunteers and their mobile devices to deploy their applications. Volunteers are people equipped with personal smartphones who accept to participate in VIVO experiments. For each experiment, volunteers can choose whether participate or not using the VIVO Client application. By means of a *Web Interface*[1], developers have the possibility to define new experiments, upload the source code of the corresponding applications, and download the collected data. Experiments uploaded to the VIVO testbed are checked and validated, during an alpha testing phase, with regard to respecting privacy and trust issues. Only accepted experiments can be deployed on volunteer devices. The alpha testing is performed during the pre-deployment phase and it checks the impact of the experiment on the overall system performance and on the user's privacy. We utilize Portable Opensource Energy Monitors (POEM) [36] to measure the energy overhead of the application and an extension of the Mockingbird platform [37] to monitor the information leakage. Mockingbird performs an on-device evaluation to retrieve the information accessible from the experiment application, e.g., when and how many times it access the file systems, the sensors, the contacts, etc. This platform produces an access-report that is compared with the experiment description in order to detect access patterns not compliant with the application task.

VIVO consists of three main components, which will be discussed in turn, namely VIVO Server, VIVO Client, and VIVO Client API.

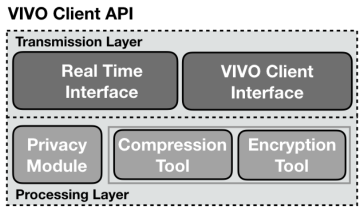
The *VIVO Server* is the main back-end platform of the architecture. It controls the creation of new experiments, the notification to volunteers, and the experiments data upload. The VIVO Server uses Google Firebase to push notifications to the volunteer devices to notify the availability of new experiments. The data collected from the volunteers are periodically sent to the VIVO Server, which handles the data upload from the devices and their storage on the *VIVO Database (VIVO DB)*. Once an experiment is terminated, the VIVO Server allows the developer to download the collected data through the web page.

VIVO is enhanced by the functionalities of the *Syndesi* IoT testbed. Syndesi [28] is a framework interconnecting heterogeneous devices from wireless sensor

[1]The VIVO Web Interface is reachable at `http://vivo.dti.supsi.ch:3000/`



(a) VIVO Client



(b) VIVO Client API

Figure 2: VIVO Client and VIVO Client API

networks as well as mobile devices, providing central resource management and user-personalized smart automation. It is implemented in the premises of the University of Geneva, although it has been designed to support portability. One of its functionalities is the gathering of environmental data, such as temperature, illuminance and humidity from the devices possessing sensors that are registered in its resource registry. The data are written into a database (Syndesi DB) hosted by the Syndesi server. The integration between VIVO and Syndesi is performed at the web service level. In particular, after terminating an experiment, the developer can download the data collected by the experiment as well as the data collected by Syndesi environmental sensors, within the experiment time window.

The *VIVO Client* enables the communication between volunteers and the VIVO Server. Volunteers contributing to the VIVO testbed are required to install and run the VIVO Client application on their devices. In particular, by means of the VIVO Client application, each volunteer can register in VIVO and run several experiments on their smartphones. The Client application displays the list of available experiments updated in real-time and allows volunteers to manage the experiment life-cycle in a very straightforward and user-friendly manner (one-click operation). As depicted in Fig. 2a, the VIVO Client acts as a middle layer between VIVO experiments and the VIVO Server. It gathers data collected by all the experiments running on the volunteer devices and manages their forwarding to the VIVO Server. All the data handled by the VIVO Client application are compressed and encrypted, as explained in Section 4.

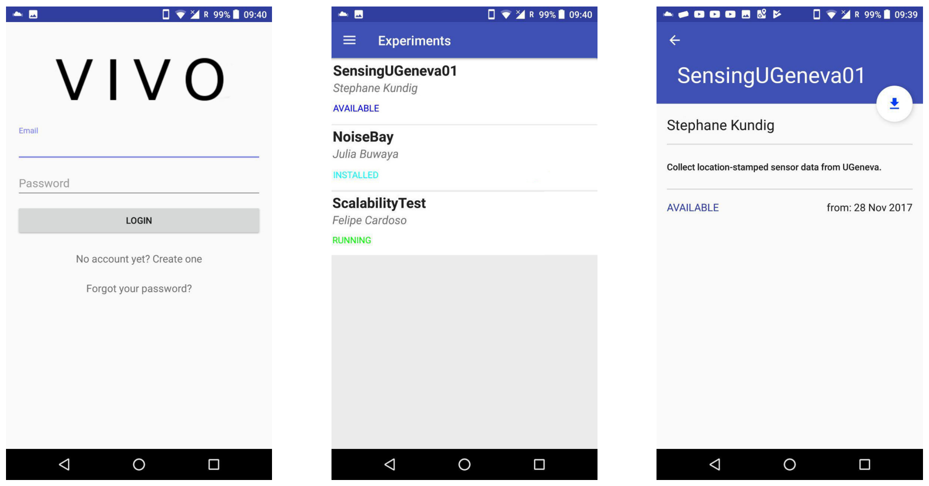
The *VIVO Client API* is a fundamental component of VIVO and enables the key features of the proposed architecture, such as security, privacy, and real-time data collection. Developers are requested to use the VIVO Client API in their application as a requirement to use VIVO and its features. The API is represented in Fig. 2b. In the Processing Layer, it provides all the tools necessary to compress (*Compression Tool*), encrypt (*Encryption Tool*), and privatize (*Privacy Module*) the collected data before the transmission from the volunteer device. As depicted in Fig. 2a, each experiment exploits the Transmission Layer of the VIVO Client API to forward the collected data to (*i*) the VIVO Server via the *VIVO Client Interface*, or to (*ii*) a Custom Server (configured by the developer) through the *Real-Time Interface*. The VIVO Client Interface is destined for offline data collection, while the Real-Time Interface, and in turn the Custom Server, enables real-time applications. In both cases, the VIVO API provides the underlying tools to encrypt, compress, and privatize the data.

4. Implementation

This section provides a detailed description of each component of the VIVO architecture, providing details on the functionalities, implementation choices, and technologies utilized in the system design. Section 4.1 describes the *VIVO Client* component, while Section 4.2 depicts the features of the *VIVO Client API*. Finally, in Section 4.3, we detail the *VIVO Server*.

4.1. VIVO Client

The *VIVO Client* is an Android application (compatible with OS version 4.2 or above) for volunteers to interact with the VIVO platform. The VIVO Client



(a) Login (b) Experiment List (c) Experiment Details

Figure 3: VIVO Client Activities

also provides synchronization of the collected data with the VIVO Server.

4.1.1. User Interface (UI)

The VIVO Client UI is the interaction point between the VIVO Server and the volunteers. The UI allows them to monitor the experiments available and running on VIVO, as well as managing their participation in each experiment.

Fig. 3 shows three screen shots related to the main foreground components of the UI. After installing the VIVO Client application, volunteers are asked to register or log-in through the Login page (Fig. 3a). Once logged in, volunteers can explore all the available experiments from the Experiment List page (Fig. 3b). Each entry in this list includes the identification parameters of the corresponding experiment (e.g., name and author), and the experiment status on the volunteer smartphone. An experiment can be: *(i) available* for download and installation on the volunteer devices; *(ii) installed* and ready for launch; *(iii) running* on the volunteer smartphone. Additional information along with a detailed description of each experiment are available on the Experiment Details page (Fig. 3c). Through this page volunteers can manage the experiment life-cycle, starting, stopping, and un-installing it any time they want to. The

VIVO Client continuously monitors the status of the experiments and displays notifications every time an experiment is created or terminated.

4.1.2. VIVO Client Data Collection

The UI has a key function in the interaction between volunteers and VIVO, nonetheless the VIVO Client performs several fundamental tasks in the background. These tasks are related to the *data collection process*. The VIVO Client is in charge of all the processing necessary for sending the experiment data from volunteer smartphone to the VIVO Server. Experiment data are sent to the VIVO Client, through the VIVO Client API, in the format of *data blocks*. A *data block* is a structure of data containing three fields: *data*, *timestamp*, and *type*. The field *data* contains an encrypted and compressed version of the data (details on data preprocessing will be given in Section 4.2.3). The *timestamp* is the time at which the data was collected, while the *type* field is used for differentiating the types of data and is defined by the developer during the experiment development phase. There are no restrictions on the type of data that can be collected (i.e., string, number, custom structure). To receive data from the running experiments, the VIVO Client instantiates a *Service* component named *Data Receiver*. This represents the VIVO Client connection with the VIVO Client API. Every time this component receives a data block from the VIVO Client API, it (*i*) verifies whether the data sender is an authorized application by checking its package name, (*ii*) extends the data block by adding a field named *experiment ID*, which identifies the experiment that generated the data, and (*iii*) temporarily stores the data block into the local database until the synchronization with the VIVO Server is performed.

To perform the synchronization we utilize the Android *Sync Adapter* component, which provides a smart way to manage data synchronization and battery consumption. Each time a synchronization is performed, a batch of data is sent to the VIVO Server. In this phase, a field named *device ID* is added to the data block to identify the device that collected the data. Every time the VIVO Server receives the data, it returns an acknowledgement and the data is deleted from the local database of the VIVO Client.

4.2. VIVO Client API

The *VIVO Client API* is a software component needed for a VIVO experiment in order to interact with the VIVO Client. An *Experiment Development Tutorial* is available on the VIVO website: it guides developers throughout the integration of existing or new Android applications with the VIVO Client API. The API is the core enabler of the VIVO architecture. Besides the interaction with the VIVO Client for forwarding the collected data, the API provides these additional features: *(i)* an interface for storing data on the VIVO Server; *(ii)* an interface for sending data to a custom server in *real-time*; *(iii)* tools for data *compression and encryption*; *(iv)* a *privacy module* to privatize the data.

The VIVO Client API is an Android Library and has the same minimum OS requirements as the VIVO Client. The API does not interact with mobile sensors and does not require any permission, thus, device heterogeneity does not affect its functionality. Thereby, developers should handle experiment dependencies by ensuring in their code whether volunteer devices meet the given requirements.

4.2.1. VIVO API Data Collection

The VIVO API Data Collection is the main function of the API, providing an interface to the VIVO Client for secure data transactions. In fact, this feature benefits from one of the main tools embedded into the API: a module for data compression and encryption. In our context, encryption is necessary because the collected data is exposed to security risks during both the API-Client and the Client-Server transmission. We address this issue by encrypting the data block locally (within the application), before sending it to the VIVO Client. To accomplish this task, the API makes use of asymmetric cryptography. As this technique can encrypt a limited block of data each time, the API compresses the data before encryption. In such a way, a larger amount of data can be encrypted in a single block. Experiment developers have to create a public-private key pair and configure the API for the usage of the public key.

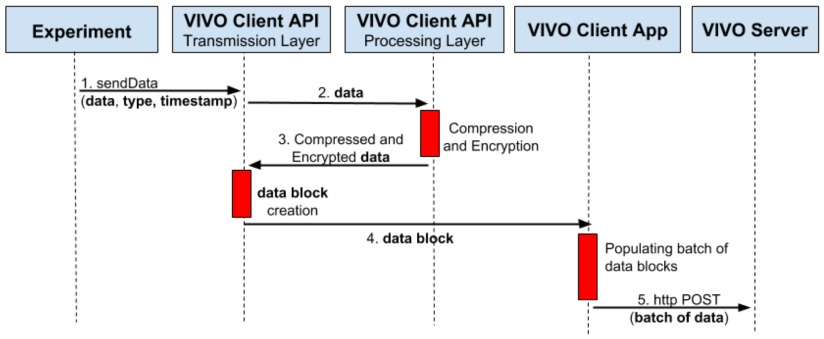

Experiment
VIVO Client API Transmission Layer
VIVO Client API Processing Layer
VIVO Client App
VIVO Server
1. sendData (**data**, **type**, **timestamp**)
2. **data**
Compression and Encryption
3. Compressed and Encrypted **data**
data block creation
4. **data block**
Populating batch of data blocks
5. http POST (**batch of data**)

Figure 4: Sequence diagram of the VIVO API (offline) data collection

Fig. 4 shows a diagram representing the sequence of actions performed by each component of the system in a data collection scenario, where *(i)* the collected data is compressed and encrypted by means of the Processing Layer of the VIVO Client API, *(ii)* the data is encapsulated in a data block in the Transmission Layer and then forwarded to the VIVO Client application, which *(iii)* accumulates data blocks in a batch of data until the synchronization with the VIVO Server is accomplished.

4.2.2. Real-Time Data Collection

Data collected in the VIVO DB is suitable for *offline* data post-processing and analysis. However, this solution does not fit real-time data processing and applications with low latency requirements. As the Sync Adapter framework of the VIVO Client does not assure real-time synchronization, it is not guaranteed that the collected data reaches the VIVO Server with a short delay. To enable real-time data collection and low-latency applications, we propose a simple interface, named Real-Time Interface, for integrating a Custom Server endpoint into the architecture. The developer should only set the Custom Server address in the VIVO API settings to perform HTTP requests with the Real-Time Interface, without configuring any server-side API. Different levels of request customization are available, ranging from a simple request with only raw data to a highly customized HTTP request. Parameters available for customization are: request body, headers, path, and callback on response.

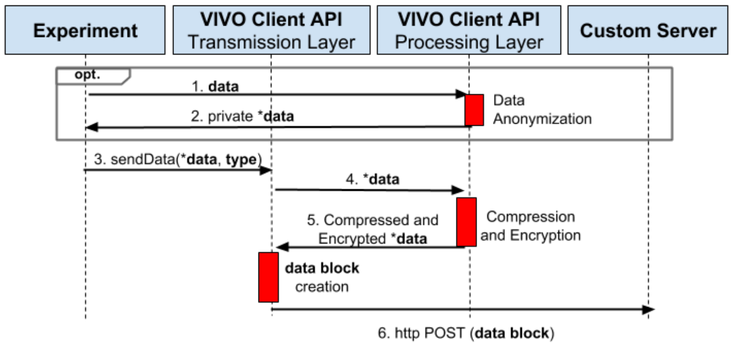

Experiment
VIVO Client API Transmission Layer
VIVO Client API Processing Layer
Custom Server
opt.
1. data
2. private *data
Data Anonymization
3. sendData(*data, type)
4. *data
Compression and Encryption
5. Compressed and Encrypted *data
data block creation
6. http POST (data block)

Figure 5: Sequence diagram of the VIVO API (real-time) data collection

Fig. 5 shows a sequence diagram representing a real-time data collection scenario. In this figure, we depict the optional sequence of actions required to perform data anonymization, which can be implemented also in the offline scenario. The Processing Layer of the VIVO Client API is in charge of anonymizing, compressing, and encrypting data before the transmission to the Custom Server. Differently from the offline scenario - where a data block is buffered in a batch of data and then transmitted according to the Sync Adapter policy - in the real-time scenario each data block is sent to the Custom Server without any buffering and additional delay.

4.2.3. Compression and Encryption Tools

Compression and encryption are fundamental tools in the proposed architecture. These features are bundled in a utility Class and use only Java standard libraries. To compress data we use the Deflate algorithm, while for the encryption we adopt asymmetric encryption implemented by the RSA algorithm. Developers can freely decide the key size for the RSA algorithm[2]. This encryption technique is highly secure if a large key size is selected but it supports a limited data block size at each encryption, which in turn depends on the key size. Thus, developers should take into account the size of the data before using

[2]We strongly suggest to use a key size of at least a 3072-bit as recommended by NSA: `https://cryptome.org/2016/01/CNSA-Suite-and-Quantum-Computing-FAQ.pdf`

the API. The maximum block size b can be computed as: $b = k - p$, where k is the key size (in byte) and p is the padding size (currently fixed at 11 byte).

4.2.4. Privacy Module

One of the key points of the proposed architecture is to ensure security and privacy at the client side. The privacy module provides routines to anonymize data by removing any personal information from identifiers collected during an experiment. First, we created a simple tool, which consists of an Anonymous ID Generator. Given a set of IDs that can potentially be used to identify a user, it produces a new set of IDs using the SHA-256 algorithm [38]. This simple anonymization is, however, generally insufficient to preserve privacy, as any personal information can be used to identify a user. For this reason, we have implemented an interface for supporting differentially private computations.

Differential privacy [39] is a property that provides an upper bound on the information a third party can obtain from the data after the release. Consider the experiment example of a simple survey composed of some yes-or-no questions. If we use a differentially private method to collect and analyze volunteer answers, then any third person that sees a statistical result over the answers (e.g., the proportion of participants saying yes) would not be able to identify the answers of individuals up to a certain specified privacy parameter called the *privacy loss*. The simplest method available is the Laplace Mechanism [40], which can be used when the statistic of interest is a real number. We also implemented the Hybrid Mechanism [41], which can be seen as an extension of the Laplace Mechanism for streaming computations, and the Randomized Response [40] mechanism, which is applicable to any type of multiple-choice question.

The differential private methods available in the API are usable - both for offline and real-time settings - for experiments that can be *reduced to a survey*. Any experiment can be *reduced to a survey*, when the collected data belongs to a bounded set of numbers that correspond to possible answers. As an example, we consider a crowd-sensing application that collects the heart rate of volunteers. This experiment can be viewed as a multiple choice survey where the sensors act as participants and the sensor measurements represent votes.

4.3. VIVO Server

The *VIVO Server* is a Node.js web application based on the *KeystoneJS* Framework. It is backed by the VIVO DB, which is a MongoDB database. This architecture provides a versatile and flexible *No-SQL* backend solution suitable for the scale of this project. The VIVO Server supports the overall architecture handling the experiments, the notification mechanism, the data collection engine, and the integration with the Syndesi IoT testbed.

4.3.1. Experiment Management

A VIVO *experiment* is an instance of an Android application that is built considering the guidelines in the development tutorial. In particular, the developer needs only to (*i*) integrate and configure the VIVO Client API as a library in the application project, and (*ii*) name the application package with a fixed string. An experiment can be deployed only once. If developers want to run an experiment multiple times, they have to create new experiments based on the same application. Once a developer has been approved by the administrator, she/he can upload applications and manage experiments from the experiment page. Every time the developer instantiates an experiment, a dedicated page is created on the web interface for experiment management. From this page the developer can request the administrator approval, start, and stop the experiment. Once the approval is granted, the experiment will be made available to all the volunteers through the VIVO Client application. Finally, when developers stop the experiment they can download (from the experiment page) all the data collected from the experiment instances installed on volunteer devices.

4.3.2. Notifications

The VIVO Server makes use of Firebase Cloud Messaging (FCM)[3] to manage notifications. FCM is a cross-platform messaging solution for delivering

[3]`https://firebase.google.com/docs/cloud-messaging/`

notification messages at a very reduced cost to drive user re-engagement. By means of FCM, the VIVO Server sends notifications to the VIVO Client applications about new and finished experiments. FCM is also integrated with the VIVO Client. Each time an FCM message arrives at the VIVO Client, a system notification is issued and displayed on the notification bar.

4.3.3. VIVO Server Data Collection

The *data collection system* manages the collected data in the VIVO Server, which periodically receives data blocks from volunteer devices. As described in Section 4.1, a single data block is composed of: *data*, *type*, *timestamp*, *experiment ID*, and *device ID*. Every time a device is synchronized with the VIVO Server, a batch of data blocks are sent in a JSON structure. A batch may contain data from different experiments. The VIVO Server dispatches each block to the correct database based on the *experiment ID*. Every time an experiment finishes, the VIVO Server creates a JSON structure that combines the experiment data, which the developers can download from the experiment page.

4.3.4. Syndesi Integration

The VIVO Server supports the integration with the *Syndesi* IoT testbed. Syndesi continuously collects environmental data from multiple sensors in a smart office environment at the University of Geneva. Developers can choose to utilize the above environmental measurements along with data collected in their experiments by enabling the “Environmental Data” option in the experiment page. This allows the developer to download, at the end of the experiment, Syndesi sensor data generated during the experiment duration. Integration of Syndesi resources with the VIVO platform is accomplished via designated APIs, which use secure HTTP connections to expose the database resources in the form of JSON files. A customized parser at the receiving end, i.e., in the VIVO Server, utilizes the above APIs to integrate the resources in Syndesi with VIVO, and provide them to the developers in a JSON format.

5. Example Scenarios

Through the VIVO testbed, some real-world applications have been already successfully implemented. Here, we overview the experiments and the usage of VIVO features in these application scenarios.

5.1. Human Behavior Data Collection (HBDC)

The *HBDC* experiments aim to collect large-scale data of human beings with the objective of understanding and predicting the subjects' behavior and the social dynamics among them. The purpose is to investigate the forces that drive people aggregation in groups (or communities) [42] and to examine the factors that mostly affect individuals' decisions and actions. Our ultimate goal is to analyze subjects' interplay for modeling social influence among them and predicting their behavior [33]. This experiment demonstrates the flexibility of VIVO in collecting heterogeneous (type of) data. In fact, for HBDC, we developed an application that collects from volunteer smartphones:

- physical information: subject position and activity detected by GPS and the Google activity recognition API, respectively;
- social information: subject social relationships revealed by contacts and call logs from the smartphone, and by social connections in Online Social Networks, such as Facebook, Twitter, and Google Plus;
- environmental information: weather based on the location of the subject, and sensor measurements from the Syndesi framework;
- personal information: subject profile information through a survey.

5.2. Indoor Localization in Environmental Crowd-Sensing (ILECS)

The *ILECS* application enables experiments to track volunteer positions in an indoor environment in real-time. We have integrated the developed smartphone indoor localization system [34] with the Syndesi framework in an application for environmental crowd-sensing. This application enables volunteers to register to the Syndesi server and contribute to its environmental monitoring scheme by sending measurements from their smartphone sensors. The sensed data are associated with an estimated indoor location before being sent back to the server. As user-location is sensitive information, we use the VIVO API to anonymize the user ID before the transmission to the VIVO Server.

In order to feed the smartphone-based indoor localization algorithm with the required inputs, the following data must be collected:

1. The RSSI from all the visible WiFi access points;
2. Smartphone on-board inertial sensor measurements;
3. Indoor floor map to constrain the estimation of the user's indoor location.

The smartphone on-board calculation combines the above information and produces online location estimates.

The application's overall functionality lifecycle is: (*i*) sensor data are queried from the smartphone sensors, e.g., temperature, illuminance, etc., depending on the smartphone model based on the polling scheme; (*ii*) location at the time of measurements is estimated via the localization algorithm; (*iii*) the sensed data are packaged along with the estimated location, the timestamp, and the user ID; (*iv*) depending on polling rate and other constraints, such as battery level and network availability, the data are synced back with the VIVO Server.

5.3. NoiseBay

In the context of the development of the VIVO platform, we launched an experiment to create a public online map of noise levels within the San Francisco Bay area using data recorded by the smartphones of private citizens[4]. The special focus of the experiment was to test load balancing and task allocation algorithms in mobile crowd-sensing applications [43], [44]. Volunteers were asked to submit non-public information about their availabilities and to download our

[4]`http://crowd.unige.ch/noiseMapSF`

NoiseBay app to collect anonymous noise levels. In the experiment, we followed a *Volunteered Geographic Information (VGI)* approach, where users are aware and actively provide data. This encourages volunteer trust towards the experiment [45] and saves the smartphone battery resources. The experiment contained several testing phases in which volunteers would either randomly collect data or were asked to collect data according to an optimized schedule. The NoiseBay app is based on the open source project *NoiseCapture* [46] and was adjusted to evaluate the applied load balancing algorithms. After the initial testing phase in San Francisco, a version of the NoiseBay app was enhanced by the compression and encryption tools available in the VIVO Client API. VIVO is especially suited for the NoiseBay app as it simplifies the distribution of the experiment and of the task schedule to volunteers. The volunteer management through the VIVO platform is an important asset if compared to an alternative distribution, e.g., via standard application download platforms.

5.4. Differential Privacy Survey (DPS)

In the *DPS* experiment, we deployed a server to create surveys and release aggregated statistics about the results while preserving differential privacy. Each survey can have many multiple choices questions. For each question, we release the number of participants who voted for a specific choice utilizing two privacy techniques (both implemented in the VIVO API): the Randomized Response and the Hybrid Mechanism. To give some insights on the effect of these techniques, we built an application survey that asks volunteers whether they like VIVO or not. We simulated 2^{18} participants, which voted YES with probability 0.6, and NO with probability 0.4. To privately compute the sum of YES votes, we employ the Hybrid Mechanism implemented in the VIVO Client API. This results in a type of *private count*. As the Hybrid Mechanism is randomised, we also make the private count consistent[5] using the transformation described in

[5] A consistent count in our example must output integer counts, and furthermore, the count must increase by either 0 or 1 after each vote.

[41]. Then, we compare the absolute error between the true count and the private one, for two settings of acceptable privacy loss $\epsilon = 1$ and $\epsilon = 0.1$. Finally, we run the hybrid mechanism 1000 times and compute the worst absolute error. The error rate measured for both privacy loss settings is quite small. The absolute error is lower than 400, whereas a simple private counting mechanism [41] would incur an absolute error proportional to 2^{18} to achieve the same privacy loss. Additionally, we noticed that the error rate is inversely proportional to the privacy loss, which means that the privacy loss should be kept to a reasonable level to make the counts useful. Nevertheless, the two privacy loss considered in our experiments provide strong guarantees. Thus, we proved that the private count mechanism provided by the VIVO API is quite useful for experiments while significantly limiting the amount of privacy that participants lose.

6. System Performance

To validate the functionality and to evaluate the performance of the proposed architecture, we developed different test-applications. In Section 6.1, we examine the scalability of VIVO by distributing an experiment to a group of volunteers scattered over the whole Switzerland. In Section 6.2, we present a comparison of the performance of the VIVO Client API with legacy solutions through a large suite of benchmarks. Finally, in Section 6.3 we compare the battery consumption of real-time upload with offline data collection.

6.1. Scalability Test

In this test, we examine the functionality and the scalability of VIVO by distributing an experiment to a group of forty volunteers scattered over the whole Switzerland. Further, this test allowed us to evaluate the robustness of VIVO by analyzing the integrity and the correctness of the collected data during the whole life-cycle, and the presence of anomalies or bugs in the implementation.

In the experiment, we gather accelerometer measurements from volunteer smartphones every minute in both offline and real-time settings. Volunteers installed the experiment from the VIVO Client, which worked without any issue. The VIVO Server handled well both the experiments and the volunteers, without any loss of data and any performance degradation. In the current version of the architecture, the VIVO Server is designed to run on a single node as a monolithic web application and, thus, it does not scale automatically on a cluster of multiple nodes. The VIVO Server instance runs on a machine with a CPU Intel(R) Pentium(R) D, dual core at 3.00GHz, 8GB DDR3 RAM, and 200GB HDD disk. To properly evaluate VIVO scalability, we should consider that the VIVO Server is a Node.js web application. Node.js operates on a single thread using non-blocking I/O calls. Thereby, it supports much more concurrent connections with respect to traditional web-serving techniques. Node architecture works well for tasks with non-intensive CPU computation, as for the VIVO Server, which performs light tasks at each synchronization. Concurrent connections capability can be computed taking into account the amount of RAM [47]. As an example, a traditional web server with 8GB of RAM can support at most few thousands of concurrent connections, while Node architecture can handle tens of thousands of simultaneous connections with the same amount of memory.

6.2. VIVO Client API Performance

To guarantee security in the data transaction, the VIVO Client API performs data compression and encryption. It is crucial to ensure that this data processing does not affect the performance and the proper operation of VIVO, both in the real-time and in the offline settings. Low latency in the data processing is mandatory for real-time applications, while a moderate battery consumption is fundamental to support volunteer involvement.

To this end, we developed two classes of experiments. First, we evaluate the delay introduced by compression and encryption at varying key size. As VIVO developers decide the size of the RSA key during the API configuration, we aim to quantify the impact of this choice in terms of additional delay. To perform these measurements, we used a *Nexus 5X* running OS version 8.1.0. We compare our proposed solution, i.e., compression and encryption, with a

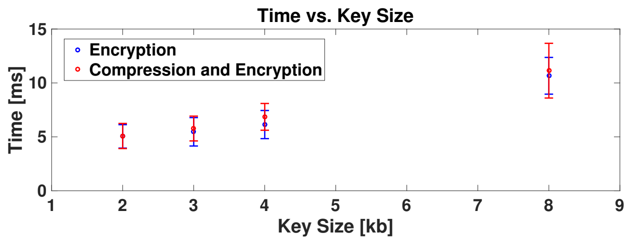
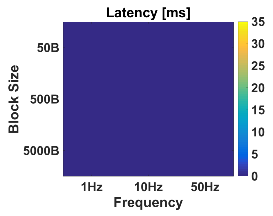


Figure 6: Data processing time vs. key size

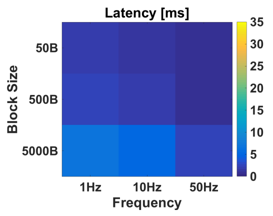
standard security solution based only on encryption. Fig. 6 represents the time measured on the VIVO Client API to perform *(i)* only encryption and *(ii)* both compression and encryption on packages of 100 byte at varying encryption key size (2048, 3072, 4096, and 8192 bits). Our proposed solution closely approaches the processing time required by the standard security solution. Data compression allows encoding information using fewer bits than the original representation, while requiring, on average, only 5% of additional time if compared to the standard security solution.

Second, we created a suite of benchmarks to measure both latency and battery consumption, comparing three processing scenarios: (*i*) raw data (does not perform any data processing), (*ii*) compression, and (*iii*) compression and encryption (VIVO Client API). We performed a set of 27 benchmarks varying the three processing scenarios, the block size (50, 500, and 5000 byte), and the frequency (1, 10, and 50 Hz). Each benchmark performs, for a given amount of time (fixed to 30 minutes), multiple data processing operations based on the frequency, which in turn determines the number of data processing operation performed in a second. We empirically choose the values of the parameters to exploit as much as possible the hardware resources at our disposal.

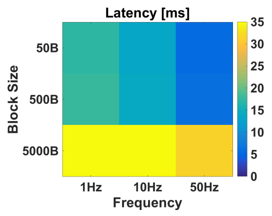
Fig. 7 compares the average latency over the processing instances of the three scenarios as a function of block size and frequency. As it can be observed in these figures, and contrarily from what we expected, for every data processing scenario, the higher the frequency the lower is the latency. Our hypothesis is that an optimization system dynamically adapts the resource allocation according to the throughput of the benchmark. We strongly believe that this optimization



(a) Raw Data

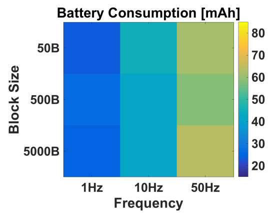


(b) Compression

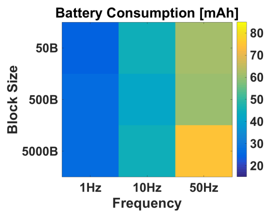


(c) VIVO Client API

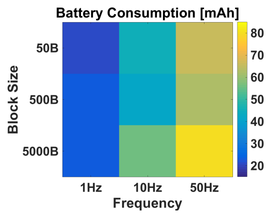
Figure 7: Latency of the three processing scenarios



(a) Raw Data



(b) Compression



(c) VIVO Client API

Figure 8: Battery consumption of the three processing scenarios

is performed by a Kernel component of the OS: the CPU Governor, which controls the CPU frequency in response to the demands of the running processes. Thereby, when the data processing frequency is low, the Governor maintains a lower CPU frequency - taking more time to process the data - with respect to the case of a higher data processing frequency. While the block size does not affect the raw data processing, in the other two scenarios we observe that the larger the block size the higher is the latency. An interesting behavior can be noticed in the VIVO Client API scenario, where encryption limits the data block size. In this experiment, we used a key size of 8192 bits, which allows to encrypt up to 1013 byte of data. Thereby, in case of larger blocks (e.g., 5000B), the API splits the data in smaller blocks introducing a computational overhead, as it can be appreciated in Fig. 7c. Finally, we observe that every parameter combinations produces an acceptable delay in every processing scenario.

The battery consumption as a function of the benchmark parameters can be seen in Fig. 8. As we expected, for every data processing scenario, the battery consumption increases with the benchmark frequency. Note in particular that

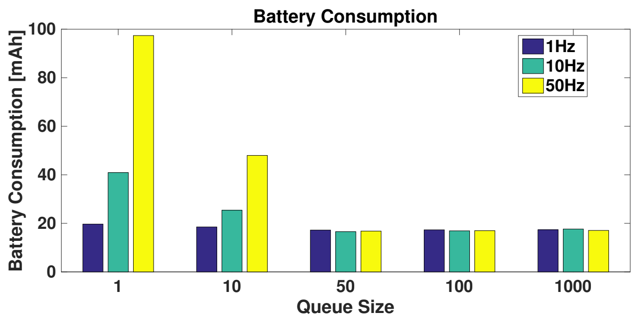


Figure 9: Battery consumption as a function of queue size and frequency

in a real experiment the frequency is set by the developer and depends on the purpose and on the requirements of the experiment itself. Further, block size does not significantly affect battery consumption in every scenario.

Overall, through these two classes of experiments, we proved that the VIVO API introduces a large suite of tools at the cost of a slightly larger latency and a moderate battery consumption if compared to legacy solutions.

6.3. Battery Consumption in Data Synchronization

In this section, we compare the offline data collection with the real-time upload in terms of battery consumption. In the former scenario, a batch of data is sent at irregular intervals based on the Sync Adapter policy, whereas in the latter, data is forwarded without any buffering. The Sync Adapter aims to transfer data while limiting the battery consumption according to the current network usage and the device sleep state. As the synchronization mechanism is strongly affected by the usage of the device, which in turn is a stochastic process, we forced the transmission every time a batch of fixed size, referred to as *queue*, is filled. In such a way the resulting consumption will be an upper bound of the real battery consumption, as the Sync Adapter manages the transmission more efficiently. In this test, we evaluate queue sizes ranging from 1 (real-time scenario) to 1000 elements. For each queue size, we performed a benchmark of 90 minutes sending data block of 50 byte at a frequency of 1, 10, and 50Hz.

Fig. 9 shows the battery consumption as a function of queue size and frequency. As it was expected, short queues consume more energy than longer ones as the system requires more frequently network infrastructure and communication. In particular, the battery consumption in case of long queues achieves almost the same value. Our hypothesis is that, in such scenarios, the battery consumption converges to a lower bound, which does not depend on the frequency and on the queue size of the data upload.

7. Conclusions

We presented the VIVO framework built onto the *enrolled crowd-sensing* model, which allows the deployment of several experiments simultaneously. VIVO also provides a paradigm-shift from (*i*) taking care of the whole experiment cycle, i.e., from experiment design up to data provision, to (*ii*) managing only the experiment application, with built-in security and privacy capabilities and the possibility to access data in real-time. We have defined and implemented VIVO architecture, and evaluated its performance. Further, we demonstrated its usability and effectiveness with four relevant real-world applications.

References

[1] B. Guo, Z. Yu, X. Zhou, D. Zhang, From participatory sensing to mobile crowd sensing, in: PERCOM Workshops, 2014 IEEE.

[2] J. Liu, H. Shen, X. Zhang, A survey of mobile crowdsensing techniques: A critical component for the internet of things, in: ICCCN, 2016, IEEE.

[3] M. Tsvetkova, T. Yasseri, E. T. Meyer, J. B. Pickering, V. Engen, P. Walland, M. Lüders, A. Følstad, G. Bravos, Understanding human-machine networks: A cross-disciplinary survey, in: ACM Comput. Surv., 2017.

[4] S. Kim, C. Robson, T. Zimmerman, J. Pierce, E. M. Haber, Creek watch: pairing usefulness and usability for successful citizen science, in: Conference on Human Factors in Computing Systems, ACM, 2011.

[5] P. Dutta, P. M. Aoki, N. Kumar, A. Mainwaring, C. Myers, W. Willett, A. Woodruff, Common sense: participatory urban sensing using a network of handheld air quality monitors, in: Sensys, ACM, 2009.

[6] R. K. Rana, C. T. Chou, S. S. Kanhere, N. Bulusu, W. Hu, Ear-phone: an end-to-end participatory urban noise mapping system, in: IPSN, 2010.

[7] P. Mohan, V. N. Padmanabhan, R. Ramjee, Nericell: rich monitoring of road and traffic conditions using mobile smartphones, in: Sensys, 2008.

[8] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, S. Madden, Cartel: a distributed mobile sensor computing system, in: Sensys, ACM, 2006.

[9] B. Pan, Y. Zheng, D. Wilkie, C. Shahabi, Crowd sensing of traffic anomalies based on human mobility and social media, in: SIGSPATIAL, ACM, 2013.

[10] Y. Chon, N. D. Lane, F. Li, H. Cha, F. Zhao, Automatically characterizing places with opportunistic crowdsensing using smartphones, in: Ubicomp, ACM, 2012.

[11] L. Bedogni, M. Di Felice, L. Bononi, By train or by car? detecting the user's motion type through smartphone sensors data, in: WD, IFIP, 2012.

[12] F. J. Villanueva, D. Villa, M. J. Santofimia, J. Barba, J. C. Lopez, Crowdsensing smart city parking monitoring, in: WF-IoT, IEEE, 2015.

[13] F. Montori, L. Bedogni, A. Di Chiappari, L. Bononi, Sensquare: A mobile crowdsensing architecture for smart cities, in: WF-IoT, IEEE, 2016.

[14] C. M. Angelopoulos, O. Evangelatos, S. Nikoletseas, T. P. Raptis, J. D. Rolim, K. Veroutis, A user-enabled testbed architecture with mobile crowdsensing support for smart, green buildings, in: ICC, IEEE, 2015.

[15] B. Guo, H. Chen, Z. Yu, X. Xie, S. Huangfu, D. Zhang, Fliermeet: a mobile crowdsensing system for cross-space public information reposting, tagging, and sharing, in: IEEE Transactions on Mobile Computing, 2015.

[16] Z. Xu, L. Mei, K.-K. R. Choo, Z. Lv, C. Hu, X. Luo, Y. Liu, Mobile crowd sensing of human-like intelligence using social sensors: A survey, in: Neurocomputing, 2018.

[17] R. K. Ganti, F. Ye, H. Lei, Mobile crowdsensing: current state and future challenges, in: IEEE Communications Magazine, 2011.

[18] J. A. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, M. B. Srivastava, Participatory sensing, 2006.

[19] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, A. T. Campbell, A survey of mobile phone sensing, in: Communications magazine, 2010.

[20] C. M. Angelopoulos, S. Nikoletseas, T. P. Raptis, J. D. Rolim, Characteristic utilities, join policies and efficient incentives in mobile crowdsensing systems, in: Wireless Days (WD), 2014 IFIP, IEEE, 2014.

[21] C. M. Angelopoulos, S. Nikoletseas, T. P. Raptis, J. Rolim, Design and evaluation of characteristic incentive mechanisms in mobile crowdsensing systems, in: Simulation Modelling Practice and Theory, Elsevier, 2015.

[22] A. Krause, E. Horvitz, A. Kansal, F. Zhao, Toward community sensing, in: Proceedings of the 7th international conference on Information processing in sensor networks, IEEE Computer Society, 2008.

[23] A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, G. Challen, Phonelab: A large programmable smartphone testbed, in: Workshop on Sensing and Big Data Mining, ACM, 2013.

[24] G. Larkou, M. Mintzis, S. Taranto, A. Konstantinidis, P. G. Andreou, D. Zeinalipour-Yazti, Demonstration abstract: Sensor mockup experiments with smartlab, in: Information Processing in Sensor Networks, IEEE, 2014.

[25] R. K. Balan, A. Misra, Y. Lee, Livelabs: Building an in-situ real-time mobile experimentation testbed, in: HotMobile, ACM, 2014.

[26] A. Striegel, S. Liu, L. Meng, C. Poellabauer, D. Hachen, O. Lizardo, Lessons learned from the netsense smartphone study, in: ACM SIGCOMM, 2013.

[27] J. Fernandes, M. Nati, N. Loumis, S. Nikoletseas, T. P. Raptis, S. Krco, A. Rankov, S. Jokic, C. M. Angelopoulos, S. Ziegler, Iot lab: Towards co-design and iot solution testing using the crowd, in: RIoT, IEEE, 2015.

[28] O. Evangelatos, K. Samarasinghe, J. Rolim, Syndesi: A framework for creating personalized smart environments using wireless sensor networks, in: Distributed Computing in Sensor Systems, IEEE, 2013.

[29] Z. Zhao, S. Kuendig, J. Carrera, B. Carron, T. Braun, J. Rolim, Indoor location for smart environments with wireless sensor and actuator networks, in: Conference on Local Computer Networks, 2017.

[30] Y. Chen, H. Chen, S. Yang, X. Gao, F. Wu, Jump-start crowdsensing: A three-layer incentive framework for mobile crowdsensing, in: IWQoS, 2017.

[31] F. Restuccia, N. Ghosh, S. Bhattacharjee, S. K. Das, T. Melodia, Quality of information in mobile crowdsensing: Survey and research challenges, in: ACM Trans. Sen. Netw., 2017.

[32] G. Han, L. Liu, S. Chan, R. Yu, Y. Yang, Hysense: A hybrid mobile crowdsensing framework for sensing opportunities compensation under dynamic coverage constraint, in: IEEE Communications Magazine, 2017.

[33] L. Luceri, T. Braun, S. Giordano, Social influence (deep) learning for human behavior prediction, in: Proceedings of CompleNet', Springer, 2018.

[34] J. L. Carrera, Z. Zhao, T. Braun, Z. Li, A. Neto, A real-time robust indoor tracking system in smartphones, in: Computer Communications, 2017.

[35] M. J. Abadi, L. Luceri, M. Hassan, C. T. Chou, M. Nicoli, A collaborative approach to heading estimation for smartphone-based pdr indoor localisation, in: IPIN, IEEE, 2014.

[36] A. Ferrari, D. Gallucci, D. Puccinelli, S. Giordano, Detecting energy leaks in android app with poem, in: PerCom Workshops, IEEE, 2015.

[37] A. Ferrari, D. Puccinelli, S. Giordano, Managing your privacy in mobile applications with mockingbird, in: PerCom Workshops, IEEE, 2015.

[38] National Institute of Standards and Technology (NIST), Fips-180-2: Secure hash standard, 2002, [www.itl.nist.gov/fipspubs].

[39] C. Dwork, Differential privacy, in: ICALP, Springer, 2006.

[40] C. Dwork, A. Roth, The algorithmic foundations of differential privacy, in: Foundations and Trends in Theoretical Computer Science, 2013.

[41] T. H. Chan, E. Shi, D. Song, Private and continual release of statistics, in: Automata, Languages and Programming, Springer, 2010.

[42] L. Luceri, A. Vancheri, T. Braun, S. Giordano, On the social influence in human behavior: Physical, homophily, and social communities, in: Conference of Complex Networks and their Applications, Springer, 2017.

[43] J. Buwaya, J. D. P. Rolim, Atomic routing mechanisms for balance of costs and quality in mobile crowdsensing systems, in: DCOSS, 2017.

[44] J. Buwaya, J. D. P. Rolim, Mobile crowdsensing from a selfish routing perspective, in: IPDPS Workshops, 2017.

[45] I. Bilogrevic, M. Ortlieb, If you put all the pieces together...: Attitudes towards data combination and sharing across services and companies, in: Conference on Human Factors in Computing Systems, ACM, 2016.

[46] G. Guillaume, A. Can, G. Petit, N. Fortin, S. Palominos, B. Gauvreau, E. Bocher, J. Picaut, Noise mapping based on participative measurements, in: Noise Mapping, 2016.

[47] M. Abernethy, Just what is node. js, in: IBM Developer Works, 2011.